L i n g u a g e m C

por

Prof. Dr. Paulo Roberto Gomes Luzzardi

FaceBook: Paulo Roberto Gomes Luzzardi
WhatsApp: 91648037
e-mail: pluzzardi@gmail.com e pluzzardi@yahoo.com.br
e-mail (Senac): prgomesluzzardi@senacrs.edu.br
DropBox: https://www.dropbox.com/sh/kea9kr4j2qttnjg/4xXvw0pvxX?m
Hostinger: http://pluzzardi.w.pw e http://pluzzardi.zz.mu (Home Page)
OS X Server (Apple): http://pluzzardi.no-ip.info:8080 (Home Page)

Co-Autor: **Ricardo Andrade Cava**

Revisão: Maio, 2003

Bibliografia:

SCHILDT, Herbert - Turbo C: Guia do Usuário, Editora: McGraw-Hill

Pelotas, domingo, 21 de setembro de 2014 (00:15:33 am)

Sumário

Lista de figuras

Lista de tabelas

1. Introdução

1.1 - Histórico

A linguagem de programação C foi desenvolvida nos anos 70 por Dennis Ritchie em um computador DEC PDP-11 que utilizava Sistema Operacional UNIX.

1.2 - Evolução

A partir de uma linguagem mais antiga chamada BCPL, desenvolvida por Martin Richards, surgiu uma linguagem chamada B, inventada por Ken Thompson que influenciou o desenvolvimento da linguagem de programação C.

1.3 - Características

- Linguagem de nível médio (combina recursos de alto e baixo nível);
- Bastante portável;
- Não é fortemente tipada;
- Permite a manipulação de bits, bytes e endereços;
- Permite escrever programas modulares e estruturados.

1.4 - Utilização

Permite o desenvolvimento e implementação de *software* básico, tais como:

- Sistemas Operacionais;
- Interpretadores;
- SGBD (**S**istema **G**erenciador de **B**anco de **D**ados);
- Editores (de texto e gráficos);
- Compiladores;
- Programas educacionais.

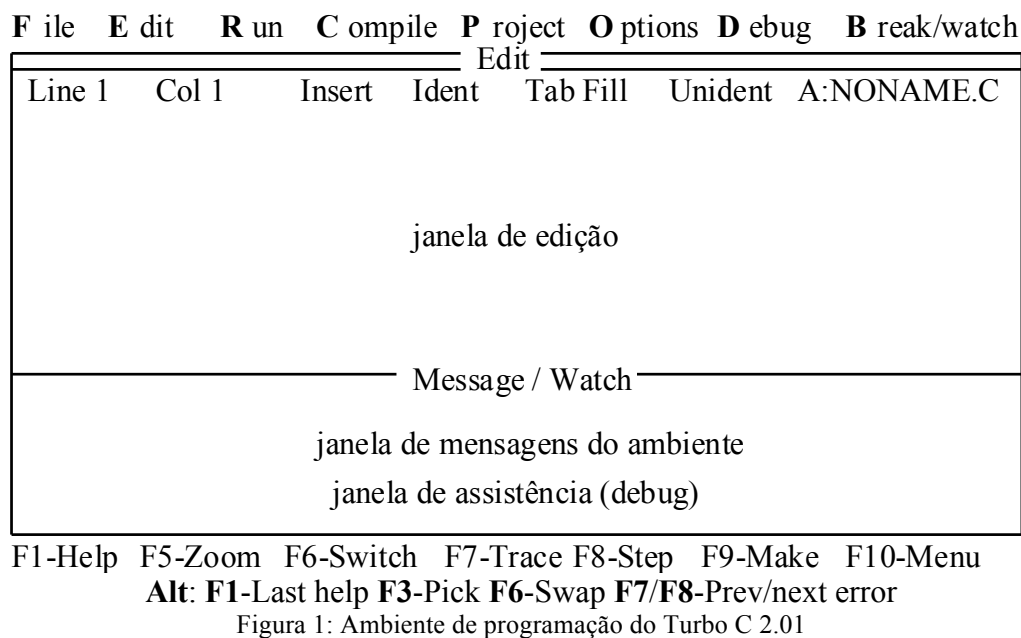
2. Ambiente Turbo C 2.01

```
F ile   E dit   R un   C ompile   P roject   O ptions   D ebug   B reak/watch
═══════════════════════════ Edit ═══════════════════════════
Line 1   Col 1   Insert   Ident   Tab Fill   Unident   A:NONAME.C

                    janela de edição

─────────────────── Message / Watch ───────────────────
          janela de mensagens do ambiente
          janela de assistência (debug)

F1-Help  F5-Zoom  F6-Switch  F7-Trace F8-Step  F9-Make  F10-Menu
Alt: F1-Last help F3-Pick F6-Swap F7/F8-Prev/next error
```

Figura 1: Ambiente de programação do Turbo C 2.01

2.1 File

A opção **File** (arquivo) permite criar um novo programa fonte, carregar um arquivo, salvar um arquivo, trocar diretório corrente, renomear arquivos, saída temporária ao DOS (***D**isk **O**peration **S**ystem* – sistema operacional em disco) e sair para o sistema operacional. O atalho pelo teclado é [**ALT**][**F**]

Load (carregar) [F3]

É exibida uma caixa de diálogo solicitando o nome do arquivo a ser carregado, se for pressionado a tecla <**enter**> é exibida uma outra caixa de diálogo contendo os nomes de todos os arquivos "*.c" contidos no disco. O programador seleciona um arquivo (seta direita, esquerda, acima ou abaixo) e pressiona <**enter**>, logo após, o arquivo é carregado do **disco** para o **editor**.

Pick (pegar)[ALT][F3]

É exibida uma lista com os nomes dos últimos arquivos que foram carregados. O programador seleciona um deles (seta acima ou abaixo) e então este é carregado do **disco** para o **editor**. Os nomes dos arquivos ficam armazenados em um arquivo (disco) chamado "**tcpick.tcp**".

New (novo)

Permite ao programador criar e editar um **novo** arquivo. Este arquivo possui o nome “**noname.c**” (sem nome), este nome deve ser alterado quando o arquivo for salvo no disco.

Save (salvar) [F2]

Salva o conteúdo do editor no disco. Se o arquivo não possuir nome ele será gravado como “**noname.c**”. Este nome pode ser renomeado. Todo arquivo deve ter um nome que não seja “**noname.c**”.

Write to (escrever como)

Escreve o conteúdo do editor num outro arquivo indicado pelo usuário (conserva o antigo no disco e o novo no editor).

Directory (diretório)

A opção **directory** mostra um diretório, de acordo com uma máscara especificada, permite ainda que um arquivo seja carregado.

Change dir (trocar de diretório)

Permite trocar o **drive** e/ou subdiretório (“**drive:\path**”) corrente.

Os Shell (sair temporariamente ao sistema operacional)

Saída temporária do Turbo C para o sistema operacional. Para retornar ao ambiente deve-se digitar “**exit**”.

Quit (sair) [ALT][X]

Saída definitiva do Turbo C. Volta ao sistema operacional.

2.2 Edit

A opção **Edit** (editar) permite a criação ou edição de programas fontes. Para sair do editor deve-se pressionar a tecla F10 ou ALT mais a letra maiúscula da opção do menu principal. O atalho pelo teclado é [**ALT**][**E**].

2.3 Run

A opção **Run** (executar) permite executar um programa (antes o programa é compilado). Possui ainda algumas opções para depudar (**debug**) o programa. O atalho de teclado é [**ALT]**[**R**].

Run (executar) [CTRL][F9]

Compila o programa, “linka” as unidades e executa o programa.

Program reset (resetar o programa) [CTRL][F2]

Termina a execução de um programa que está sendo executado com o **debug**, ou seja, desmarca a linha atual de execução do **debug**.

Go to cursor (ir para o posição do cursor) [F4]

Força o **debug** a executar todas as linhas até a posição do cursor, ou seja, pula todas as linhas intermediárias, executando-as.

Trace into (traçar dentro de uma função) [F7]

Força o **debug** a executar linha a linha do programa, "entrando" (executando) dentro das funções definidas pelo programador.

Step over (passar sobre a função) [F8]

Força o **debug** a executar linha a linha sem entrar nas funções definidas pelo programador.

User screen (visualizar a tela de usuário) [ALT][F5]

Exibe a tela de execução, ou seja, mostra o resultado do programa.

2.4 Compile

A opção **Compile** (compilar) possui os itens de compilação e configuração da compilação. O atalho de teclado é [**ALT**][**C**].

Compile to OBJ (compilar) [ALT][F9]

Compila o programa fonte e “linka” as unidades gerando o arquivo ".OBJ". O nome do programa objeto é o mesmo do programa fonte ou do “**Primary File**” + ".OBJ".

Make EXE file (fazer) [F9]

Compila o arquivo primário (“**Primary File**: “) ou o arquivo presente no editor. Todos os arquivos que dependem deste são compilados, inclusive as

unidades que foram alteradas gerando o arquivo ".EXE". O nome do programa executável é o mesmo do programa fonte ou do “**Primary File**” + ".EXE".

Link EXE file (ligar)

Obriga a “link-edição” (ligação) dos arquivos secundários e bibliotecas ".LIB" e ".OBJ" ao arquivo primário.

Build all (construir)

Recompila todos os arquivos relacionados ao arquivo primário (“**Primary File**”).

Primary C file: (arquivo principal)

Arquivo principal, utilizado pelas opções **Make**, **Build** e **Link**.

Get info (mostra informações)

Exibe uma janela contendo informações do arquivo fonte.

2.5 Project

A opção **Project** (projeto) possui os itens relativos ao projeto. O atalho de teclado é [**ALT**][**P**].

Project name: (nome do projeto)

Nome do arquivo de projeto (principal), ou seja, o arquivo ".OBJ" e ".EXE" terão o nome do projeto. Este arquivo contém os nomes de todos os fontes e/ou objetos que serão “linkados” ao arquivo principal. Possui a extensão ".PRJ".

Break make on: (parada da opção Make)

Força a parada da opção **Make** quando:

- **Warnings**: Houver advertências;
- **Errors**: Houver erros;
- **Fatal errors**: Houver erros fatais;
- **Link**: Antes de “Linkar”.

Auto dependencies: (auto dependência)

- **ON**: Verifica as dependências de todos os arquivos ".OBJ" dentro do **Project**.

- **OFF**: Não verifica.

Clear project: (apagar projeto)

Permite deletar o nome do projeto colocado na opção “**Project name**”.

Remove messages: (remove mensagens)

Apaga ou não as mensagens de erros da janela de mensagens.

2.6 Options

Possui alguns itens de configuração. O atalho de teclado é [**ALT**][**O**].

Compiler: (compilador)

Permite a inserção de diretivas de compilação sem escrevê-las no programa-fonte.

- **Model**: Modelo de memória. Por *default* o modelo de memória é **small**.

    - **Tiny**: Menor de todos modelos de memória. Existe 64K bytes para programa, dados e vetores. Utiliza-se ponteiro **near**. Este programa pode ser convertido para ".COM".
    - **Small**: 64K bytes para dados e 64K bytes para código. Utiliza-se ponteiro **near**. Bom tamanho para aplicações médias.
    - **Medium**: 64K bytes para dados e 1M bytes para código. Utiliza-se ponteiro **far** para código e não para dados. Este é o melhor modelo para programas grandes, mas não armazena muitos dados na memória.
    - **Compact**: Inverso do **medium**. 64k bytes para código e 1M bytes para dados. Utiliza-se ponteiro **far** para dados. Este é melhor quando seu código é pequeno e você precisa endereçar uma porção de dados.
    - **Large**: 1M bytes para código e dados. Utiliza-se ponteiro **far** para ambos. É necessário somente em aplicações muito grandes.
    - **Huge**: Dados estáticos ocupam mais do que 64k bytes. Utiliza-se ponteiro **far** para código e dados.

- **Code Generation** (geração de código)

    - *Calling comention* (C ou Pascal):
    - *Instruction set* (8088/8086 ou 80186/80286):
    - *Floating point* (*Emulation*, 8087/80287 ou *none*):
    - *Default char type* (*Signed* ou *Unsigned*):
    - *Alignment* (*Byte* ou *Word*):
    - *Generate underbar* (ON ou OFF):

- *Merge duplicate string* (ON ou OFF):
- *Standard stack frame* (ON ou OFF):
- *Test stack overflow* (ON ou OFF):
- *Line Numbers* (ON ou OFF):
- *OBJ debug information* (ON ou OFF):

- **Optimization** (otimização)

  - *Optimize for* (*Speed* ou *Size*):
  - *Use register variables* (ON ou OFF):
  - *Register optimization* (ON ou OFF):
  - *Jump optimization* (ON ou OFF):

- **Source** (fonte)

  - *Identifier length* (1 à 32):
  - *Nested coments* (ON ou OFF):
  - *ANSI keywords only* (ON ou OFF):

- **Errors** (erros)

  - ***Errors***: *Stop after* (0 à 255): Aborta após "n" erros
  - ***Warnings***: *Stop after* (0 à 255): Aborta após "n" advertências
  - ***Display warnings*** (ON ou OFF): Exibe ou não as advertências
  - ***Portability warnings*** (advertências de portabilidade):

    - A: *Non-portable point conversion* (ON ou OFF):
    - B: *Non-portable point assigment* (ON ou OFF):
    - C: *Non-portable point comparison* (ON ou OFF):
    - D: *Constant out of range in comparison* (ON ou OFF):
    - E: *Constant is long* (ON ou OFF):
    - F: *Conversation may lose significant digits* (ON ou OFF):
    - G: *Mixing pointers to signed and unsigned char* (ON ou OFF):

- **ANSI violations** (violações ao padrão ANSI)

A: *'ident' not part of structure* (ON ou OFF):
B: *Zero length structure* (ON ou OFF):
C: *Void function may not return a value* (ON ou OFF):
D: *Both return and return of a value used* (ONouOFF):
E: *Suspicius pointer conversation* (ON ou OFF):
F: *Undefined structure 'ident'* (ON ou OFF):
G: *Redefinition of 'ident' is not identical* (ONouOFF):
H: *Hexadecimal or octal constant too large* (ON ou OFF):

- **Common errors** (erros comuns)

  A: *Function should return a value* (ON ou OFF):
  B: *Unreachable code* (ON ou OFF):
  C: *Code has no effect* (ON ou OFF):
  D: *Possible use of 'ident' before definition* (ON ou OFF):
  E: *'ident'is assigned value which is never used* (ON ou OFF):
  F: *Parameeter 'ident' is never used* (ON ou OFF):
  G: *Possibly incorrect assigment* (ON ou OFF):

- **Less common errors** (erros menos comuns)

  A: *Superfuous & with function or array* (ON ou OFF):
  B: *'ident' declared but never used* (ON ou OFF):
  C: *Ambiguous operators need value* (ON ou OFF):
  D: *Structure passed by value* (ON ou OFF):
  E: *No declaration for functions 'ident'* (ON ou OFF):
  F: *Call to function with no prototype* (ON ou OFF):

- **Names** (nomes)

  - *Code names* (*Segment name**, *Group names** ou *Class names**)
  - *Data names* (*Segment name**, *Group names** ou *Class names**)
  - *BSS names* (*Segment name**, *Group names** ou *Class names**)

Linker (ligador)

- *Map file* (ON ou OFF):
- *Initialize segments* (ON ou OFF):
- *Default libraries* (ON ou OFF):
- *Graphics library* (ON ou OFF):
- *Warm duplicate symbols* (ON ou OFF):
- *Stack warning* (ON ou OFF):
- *Case-sensitive link* (ON ou OFF):

Environment (ambiente)

Permite configurar o ambiente de trabalho.

- ***Message tracking*** (*All Files*, *Current Files* ou *Off*):
- ***Keep messages*** (*No* ou *Yes*):
- ***Config auto save*** (ON ou OFF): Se ligado, atualiza o arquivo de configuração (“**tcconfig.tc**”) quando o programador sai do Turbo C.

- ***Edit auto save*** (ON ou OFF): Salva automaticamente o arquivo fonte quando o arquivo for executado (**Run**) ou quando da saída para o sistema operacional (**quit** ou **Os shell**).
- ***Backup files*** (ON ou OFF): Gera arquivo ".BAK" quando o arquivo fonte for salvo.
- ***Tab size 8***: Especifica o número de espaços da tabulação horizontal (tecla **Tab**) do editor (2 até 16 brancos).
- ***Zoom windows*** (ON ou OFF): se ligada, as janelas **Edit**, **Watch** ou **Message** estão expandidas.
- ***Screen size*** (25 *line display* ou 43/50 *line display*): Permite selecionar o número de linhas na tela: 25 placa CGA, 43 placa EGA e 50 placa VGA.

Directories (diretório)

Permite especificar "**drive:\path**" para diretórios dos arquivos utilizados e que serão gerados pelo Turbo C (por exemplo, programas fontes *.c).

- ***Include directories***: permite alterar **drive:\path** indicando a localização dos arquivos de *includes*, ou seja, ".H", normalmente, "**c:\tc\include**".
- ***Library directories***: permite alterar **drive\path** indicando a localização dos arquivos ".LIB" e ".OBJ", normalmente, "**c:\tc\lib**".
- ***Output directory*****:** permite alterar **drive\path** indicando diretório de saída. Todos os arquivos lidos ou gerados serão armazenados neste diretório.
- ***Turbo C directory***: permite **drive\path** indicando o diretório onde o Turbo C está instalado.
- ***Pick File Name*****:** permite altera **drive\path\nome** do arquivo de seleção ".PCK".
- ***Current Pick File***: Indica o **drive**, caminho e nome do arquivo de seleção corrente.
- ***Arguments*** (Argumentos): Permite a especificação de parâmetros quando um programa for executado na memória, ou seja, os argumentos da linha de comando aceitos pelo **argc** e **argv**.
- ***Save options***: Salva o arquivo de configuração "**.tc**" (*default* "**tcconfig.tc**").
- ***Retrieve options*****:** Carrega o arquivo de configuração "**.tc**" (*default* "**tcconfig.tc**").

2.7 Debug

A opção **debug** (depurador) possui os itens relacionados a depuração de um programa. O atalho de teclado é [**ALT**][**D**].

- ***Evaluate*** [CTRL][F4]: Exibe uma janela de avaliação com três opções: (*Evaluate*, *Result* e *New name*)
  - ***Evaluate***: Permite identificar uma expressão ou uma variável a ser submetida ao *Debug*.

- ***Result***: É exibido o resultado da avaliação da expressão ou variável selecionada acima.
   - ***New name***: Permite atribuir novo valor.
- ***Call stack*** [CTRL][F3]: Durante a depuração, este ítem permite chamar uma janela da pilha que contém uma lista de "funções" que mostra a posição atual, mostrando também, os parâmetros de cada chamada.
- ***Find function***: posiciona o cursor no início de uma função.
- ***Refresh display***: Retorna para a tela do ambiente.
- ***Display swapping*** (*None*, *Smart* ou *Always*): Permite estabelecer modos de visualização de *outputs* de tela durante a depuração do programa.
   - ***None***: não mostra a tela de saída do programa.
   - ***Smart***: mostra os efeitos da tela somente quando houver um comando saída.
   - ***Always***: mostra sempre a tela resultante.
- ***Source debugging*** *(ON, Standalone ou None):*

2.8 Break/watch

Na opção **Break/watch** pode-se adicionar, deletar, editar, remover todos os **watches** (assistidos) ou colocar, retirar procurar **breakpoints** (pontos de parada). O atalho de teclado é [**ALT**][**B**].

- ***Add watch*** [CTRL][F7]: Permite que a variável sob o cursor seja exibida na janela de assistência quando o *debug* for executado.
- ***Delete watch***: Permite que uma variável da janela de assistência seja deletada.
- ***Edit watch***: Permite que uma variável da janela de assistência seja editada.
- ***Remove all watches***: Remove todas as variáveis da janela de assistência.
- ***Toggle breakpoint*** [CTRL][F8]: Permite que **breakpoints** (pontos de parada) sejam colocados ou retirados.
- ***Clear all breakpoints***: Permite que todos os **breakpoints** sejam removidos.
- ***View next breakpoint***: Permite visualizar o próximo **breakpoint**.

2.9 Como usar o Debug

1º Passo: Ter as duas janelas na tela: janela de edição e janela de assistência [**F5**];

2º Passo: Marcar um ***breakpoint*** (ponto de parada) [**CTRL**][**F8**] ou opção ***Toggle breakpoint*** do menu ***Break/watch***;

3º Passo: Rodar o programa ***Run*** ou [**CTRL**][**F9**], o programa é executado até a linha anterior ao ***breakpoint***;

Observação: Para visualizar o resultado obtido na tela de execução [**ALT**][**F5**] ou a opção ***User screen*** do menu ***Run***;

4º Passo: Para selecionar as variáveis que se deseja assistir, posiciona-se o cursor sobre uma variável e [**CTRL**][**F7**] ou a opção ***Add watch*** do menu ***Break/watch***, após aparecer uma janela com a variável dentro (podendo-se alterar ou mudar a variável) pressiona-se [**RETURN**];

5º Passo: Para visualizar a execução do programa linha à linha pressiona-se [**F8**] (opção ***Step over*** do menu ***Run***) ou [**F7**] (opção ***Trace into*** do menu ***Run***):

[F8]: executa o programa linha à linha sem entrar nas funções;
[F7]: executa o programa linha à linha entrando também nas funções;

6º Passo: Pode-se ainda ***Delete watch*** (deletar variável), ***Edit watch*** (editar variável) ou ***Remove all watches*** (remover todas as variáveis) no menu ***Break/watch***.

7º Passo: Pode-se ainda desviar a execução do ***debug*** para a linha em que o cursor está [**F4**] ou a opção ***Goto cursor*** do menu ***Run***;

8º Passo: Para encerrar a execução do ***debug*** [**CTRL**][**F2**] ou a opção ***Program reset*** do menu ***Run***, deve-se ainda desmarcar todos os ***breakpoints*** através da opção ***Clear all breapoints*** do menu ***Break/watch***.

2.10 - Teclas de funções

Podem ser acessadas em qualquer ponto do ambiente de programação:

F1: Ativa o *help*
F2: Salva (*Save*) o arquivo corrente no editor
F3: Carrega (*Load*) ao arquivo
F5: Aproxima (*Zoom*) a janela ativa
F6: Troca (*Switch*) a janela ativa
F7: Executa uma linha na depuração (*Debug* - entra na função))
F8: Executa uma linha na depuração (*Debug* - não entra na função)
F9: Compila e liga o programa (*Make*)
F10: Ativa o menu principal

Alt F1: Traz de volta a última tela de ajuda
Alt F3: *Pick* (carrega um dos 8 últimos arquivos lidos)
Alt F9: Compila o arquivo no editor para OBJ

Alt F: *File*
Alt R: *Run*
Alt C: *Compile*
Alt P: *Project*
Alt O: *Options*
Alt D: *Debug*
Alt B: *Break/watch*

Alt X: Sai para o DOS
SHIFT F10 : Exibe o número da versão

2.11 - Comandos do editor

2.11.1 - Movimento básico do cursor

Seta esquerda ou **CTRL-S**	Caracter à esquerda
Seta direita ou **CTRL-D**	Caracter à direita
CTRL-Seta esquerda ou **CTRL-A**	Palavra à esquerda
CTRL-Seta direita ou **CTRL-F**	Palavra à direita
Seta cima ou **CTRL-E**	Linha acima
Seta baixo ou **CTRL-X**	Linha abaixo
CTRL-W	Deslocar texto para cima
CTRL-Z	Deslocar texto para baixo
PGUP ou **CTRL-R**	Página para cima
PGDN ou **CTRL-C**	Página para baixo

2.11.2 - Movimento rápido do cursor

HOME ou **CTRL-QS**	Vai para início da linha
END ou **CTRL-QD**	Vai para fim da linha
CTRL-HOME ou **CTRL-QE**	Vai para começo da tela
CTRL-END ou **CTRL-QX**	Vai para fim da tela
CTRL-PGUP ou **CTRL-QR**	Vai para começo do arquivo
CTRL-PGDN ou **CTRL-QC**	Vai para fim do arquivo
CTRL-QB	Vai para começo do bloco
CTRL-QK	Vai para fim do bloco
CTRL-QP	Vai para última posição do cursor

2.11.3 - Inserção e deleção

INS ou **CTRL-V**	Liga e desliga modo de inserção
CTRL-N	Insere uma linha em branco
CTRL-Y	Apaga toda linha
CTRL-QY	Apaga até o fim da linha
CTRL-H ou **Backspace**	Apaga o caracter à esquerda
DEL ou **CTRL-G**	Apaga caracter sob cursor
CTRL-T	Apaga palavra à direita

2.11.4 - Manipulação de blocos

CTRL-KB	Marca início do bloco
CTRL-KK	Marca fim do bloco
CTRL-KT	Marca uma palavra
CTRL-KC	Copia um bloco
CTRL-KY	Elimina um bloco
CTRL-KH	Esconde/mostra marcas de um bloco
CTRL-KV	Desloca um bloco
CTRL-KR	Lê um bloco do disco
CTRL-KW	Grava um bloco no disco
CTRL-KI	Move bloco para direita
CTRL-KU	Move bloco para esquerda

2.11.5 - Comandos variados

CTRL-U	Aborta qualquer operação
CTRL-OI	Auto-indentação (on/off)
CTRL-P	Prefixo do caracter de controle
CTRL-QF	Procura (*Find*)
CTRL-QA	Procura e substitui (*Find*)

B	Início do texto até o cursor
G	Texto inteiro
n	Procura a énesima ocorrência

N	Não pede confirmação
U	Maiúsculas e minúsculas iguais
W	Somente palavras inteiras

CTRL-Qn	Procura uma marca
CTRL-KD ou **CTRL-KQ**	Sai do editor, não salva
CTRL-L	Repete procura
CTRL-QL	Recupera linha
CTRL-KS	Salva e edita
CTRL-Kn	Coloca uma marca
CTRL-I ou **TAB**	Tabulação
CTRL-OT	Liga e desliga modo TAB

2.11.6 – Ajuda on-line

F1	*Help*
CTRL-F1	*Help* por comando

Funcionamento: Coloque o cursor sobre o comando (ou função) que deseja ajuda *on-line* e digite CTRL + F1.

3. Estrutura de um programa em C

3.1 - Identificadores

São os nomes criados pelo programador para fazer referência a **variáveis**, **contantes**, **funções** e **rótulos**.

Regras para a criação de identificadores (Turbo C 2.01):

- O primeiro caracter deve ser uma letra ou sublinha (_);
- Os caracteres seguintes devem ser letras, números ou sublinhas;
- Os primeiros 32 (*default*) caracteres são significativos;
- Não é permitido a utilização de caracteres em branco (caracter espaço).

Exemplos em variáveis:

```
int num_dentes = 32;
float inflação;
char a, _a;
```

Exemplos em constantes:

```
#define ESC     27
#define ENTER  13
```

Exemplos em funções:

```
x = raiz_quadrada(y);
printf("Valor: %.2f\n", inverso(n));  /* definida pelo programador */
```

3.2 – Comentários do programador

Os comentários do programador são linhas de código que não são compiladas pelo compilador, ou seja, servem apenas como anotações para serem lembradas mais tarde (por exemplo, quando forem feitas manutenções no programa). Em C os comentários podem ser feitos da seguinte maneira:

```
/* Isto é um comentário em Turbo C */
// Isto é um comentário no Borland C
```

Um comentário, pode ainda, utilizar várias linhas do programa. Veja o exemplo abaixo:

```
/* ----------------------------------------------------
   Função: STRING
   Parâmetros de entrada: x, y
   Parâmetros de saída: c, t
   Retorno: Sem Retorno
------------------------------------------------------- */
```

3.3 - Regras gerais para escrever um programa em C

Um programa em C é constituído de uma ou mais funções delimitadas por chaves { }, onde uma destas funções, obrigatoriamente é chamada **main**(). As principais regras são:

- Letras maiúsculas e minúsculas são tratadas como caracteres diferentes;
- O formato do texto é livre;
- A função **main**() especifica onde o programa começa e termina de ser executado;
- Todos os comandos são terminados por ponto e vírgula;
- Todas as variáveis devem ser declaradas;
- { função começa a ser executada;
- } função termina de ser executada.

Programa exemplo (1): Imprimir a data no seguinte formato: [Data: dd/mm/aaaa].

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int dia, mes, ano;                    /* variáveis locais utilizadas somente na função main */

 clrscr();                             /* função limpa a tela (clear screen) */
 dia = 23;
 mes = 5;
 ano = 2003;
 printf("Data: %02d/%02d/%04d\n", dia, mes, ano);
 getch();
}
```

3.4 - Palavras reservadas

Definidas por K&R (Kernighan & Ritchie)				ANSI	Turbo C	
auto	double if	static	const	asm	pascal	
break	else	int	struct	enum	_ss	_es
case	entry	long	switch	signed	interrupt	huge
char	extern	register	typedef	void	_cs	
continue	float	return	union	volatile	decl	
default	for	sizeof	unsigned	near		
do	goto	short	while	_ds		

Observação: As palavras reservadas não podem ser utilizadas pelo programador como nome de variáveis, constantes ou funções, ou seja, não servem como identificadores.

3.5 - Declaração de variáveis

Sintaxe: tipo_dado_base lista_de_variáveis;

tipo_dado_base: deve ser um tipo de dado válido (***int, char, float, double ou void***)
lista_de_variáveis: um ou mais identificadores separados por vírgula.

Exemplo:

```
void main (void)
{
 int i, j ,k;
 float a, b;
 char ch;
```

3.5.1 - Onde as variáveis podem ser declaradas

- Definidas fora de todas as funções, incluindo a função **main**() são chamadas de **variáveis globais** e podem ser acessadas em qualquer parte do programa. Estas variáveis são alocadas estaticamente na memória RAM (***R****andom* ***A****ccess* ***M****emory* – Memória de acesso randômico).
- Definidas dentro de uma função são chamadas de **variáveis locais** e só podem ser acessadas dentro desta função. Estas variáveis são alocadas dinamicamente na memória RAM. Depois que uma função é executada, estas variáveis são desalocadas.
- Na declaração de parâmetros formais de uma função. Sendo estas **locais** e alocadas dinamicamente na memória RAM.

Observação: Memória ROM (***R****ead* ***O****nly* ***M****emory* – Memória somente de leitura).

Alocação de memória: Reserva de espaço de memória (RAM) para alocar uma variável.

- **Alocação estática de memória**: Tipo de alocação de memória em que uma variável é alocada (tem um espaço reservado) na memória RAM durante toda a execução do programa. Este espaço de memória é desalocado somente quando o programa acaba.
- **Alocação dinâmica de memória**: Tipo de alocação de memória em que uma variável é alocada (tem um espaço reservado) na memória RAM temporariamen-te. Este espaço de memória é desalocado quando o espaço não é mais necessário.

Programa exemplo (2): O programa realiza uma operação de potência X^y.

```
#include <stdio.h>          /* o arquivo stdio.h é inserido dentro deste programa */
#include <conio.h>          /* o arquivo conio.h é inserido dentro deste programa */
```

```
float POT (float base, float expoente);     /* protótipo da função POT */
float resultado;                            /* variável global */

void main (void)
{
 float base, expoente;                      /* definição das variáveis locais */

 clrscr();
 printf("Base: ");                          /* função que permite imprimir dados na tela */
 scanf("%f",&base);                         /* função que permite a entrada de dados via teclado */
 printf("Expoente: ");
 scanf("%f",&expoente);
 resultado = POT(base, expoente);           /* chamada da função POT */
 printf("Resposta = %7.2f\n", resultado);
 getch();
}

float POT (float x, float y)        /* corpo da função POT definida pelo programador */
{                                   /* os parâmetros x e y são variáveis locais */
 float resp;                        /* definição das variáveis locais */

 resp = exp(log(x) * y);
 return(resp);                      /* retorno da função */
}
```

Variáveis globais: resultado, POT
Variáveis locais: base, expoente, resp, x, y

3.5.2 - Inicialização de variáveis

Em C, é possível fornecer valores iniciais a maioria das variáveis ao mesmo tempo em que elas são declaradas, colocando um sinal de igual e uma constante após o nome da variável.

```
tipo_dado_base  nome_da_variável = constante;
```

Exemplos:

```
char ch = 'a';          /* tipo_dado_base  nome_da_variável = constante_caracter */
char s = “UCPel”;       /* tipo_dado_base  nome_da_variável = constante_string */
int n = 0;              /* tipo_dado_base  nome_da_variável = constante_inteira */
float y = 123.45;       /* tipo_dado_base  nome_da_variável = constante_real */
```

3.6 - Constantes

Valores fixos que o programa não pode alterar. As constantes podem ser de qualquer tipo básico.

Tipo	**Exemplos de constantes**			
char	'a'	'n'	'9'	
int	1	123	2100	-234

float	123.23 4.34e-3	
char *	"Turbo C 2.2"	(... forma de definir uma string)

3.6.1 - Constantes hexadecimais e octais

A linguaguem de programação C permite especificar constantes inteiras em hexadecimal ou octal. Uma constante **octal** começa com um **0** (zero), enquanto que uma **hexadecimal** começa por **0x**.

Exemplos:

```
void main (void)
{
 int hexadecimal = 0xFF;          /* 255 em decimal */
 int octal = 011;                 /* 9 em decimal */
```

Observações:

- Qualquer número **octal** é formado por oito números (0 .. 7).
- Qualquer número **hexadecimal** é formado por dezesseis números (0 ..9, A, B, C, D, E, F).

3.6.2 - Constantes strings

Uma **string** é um conjunto de caracteres delimitados por aspas duplas. Em C não existe um tipo de dado especifico para definir uma **string**.

Uma **string** é definida como um vetor (unidimensional) de caracteres. Toda **string** deve ser finalizada pelo caracter especial **'\0'** (chamado de **NULL**).

Exemplo:

char s[6] = "UCPel"; ou char s[6] = {'U', 'C', 'P', 'e', 'l', NULL};

'\0' é igual a NULL

0	1	2	3	4	5
'U'	'C'	'P'	'e'	'l'	NULL

Figura 2: Vetor de caracteres

Memória ocupada: 6 bytes

Observação: 'a' é diferente de "a"

'a' ocupa 1 byte na memória RAM (é do tipo **char**)
"a" ocupa 2 bytes na memória (toda **string** é terminada com o caracter '\0')

A palavra NULL (quer dizer nulo) está definida dentro do arquivo de **header** (cabeçalho) **stdio.h** (**st**an**d**ard **i**nput **o**utput).

#define NULL '\0'

3.6.3 - Constantes especiais

As constantes especiais são utilizadas para representar caracteres que não podem ser inseridos pelo teclado. São elas:

Tabela 1: Constantes especiais

Constante	Significado
‘\b’	Retrocesso
‘\f’	Alimentação de formulário
‘\n’	Nova linha
‘\r’	Retorno de carro <CR>
‘\t’	Tab horizontal
‘\”’	Aspas duplas
‘\’’	Aspas simples
‘\o’	Zero
‘\\’	Barra invertida
‘\a’	Alerta
‘\o’	Constante octal
‘\x’	Constante hexadecimal

3.7 - Comandos do pré-processador do C

3.7.1 - O comando #define

O comando #**define** é utilizado para definir uma macro-substitução. O compilador substitui o identificador pelo valor cada vez que aquele for encontrado no programa fonte antes da compilação do programa.

#define identificador valor

Exemplos:

```
#define TRUE  !0            /* ou   #define TRUE    1    */
#define FALSE 0

#define UP     72
#define DOWN 80
#defiine LEFT  75
#define RIGHT 77

#define ENTER    13
#define ESC       27
```

3.7.2 - O comando #include

O comando #**include** faz o compilador incluir um arquivo-fonte dentro de outro arquivo fonte.

#include <header .h>   /* arquivo de header padrão do C */

ou
#include "header .h" /* arquivo de header escrito pelo programador */

Observações: Normalmente os arquivos de **header** estão localizados em *"***c:\tc\include***"*. E os arquivos de **header** do programador estão localizados em seu diretório corrente *"***c:\tc\fontes***"*.

Exemplos:

```
#include <stdio.h>        /* arquivo de header padrão do Turbo C (c:\tc\include) */
#include "luzzardi.h"     /* arquivo de header definido pelo programador (diretório corrente) */
```

4. Tipo de dados

4.1 - Tipos básicos

A tabela abaixo exibe os cinco (5) tipos de dados básicos que podem ser utilizados pelo programador para definir suas variáveis. São exibidos os **tipos** básicos, a quantidade de **bits**, a **faixa de valores** válida e o número de **bytes** que cada tipo de dados ocupa na memória RAM (memória principal) ou em disco (quando armazenados na memória secundária).

Tabela 2: Tipos de dados

Tipo	Bits	Faixa de valores	Bytes
char	8	-128 à 127	1
Int	16	-32768 à 32767	2
float	32	-3.4E-38 à 3.4E+38	4
double	64	-1.7E-308 à 1.7E+308	8
void	0	Sem valor	0

4.2 Modificadores de tipo

Os modificadores de tipo são utilizados para modificar os tipos de dados base, ou seja, se adaptando às necessidades do programador. Os modificadores modificam a quantidade de bits e bytes dos tipos-base, alterando, desta forma, a faixa de valores destes novos tipos de dados.

Tabela dos modificadores de tipos:

Tabela 3: Modificadores de tipos

Modificador de tipo	Modificação	Descrição
signed	c/sinal	Números positivos e negativos
unsigned	s/sinal	Números positivos
long	longo	Aumenta o número de bytes do tipo
short	curto	Diminui o número de bytes do tipo

Tabela de tipos básicos modificados:

Tabela 4: Tipos de dados básicos modificados

Tipo	Bits	Faixa de valores	Bytes
unsigned char	8	0 à 255	1
signed char	8	-128 à 127	1
unsigned int	16	0 à 65535	2
signed int	16	-32768 à 32767	2
short int	16	-32768 à 32767	2
long int	32	-2147483648 à 21474483647	4
unsigned short int	16	0 à 65535	2
signed short int	16	-32768 à 32767	2
unsigned long int	32	0 à 4294967295	4
signed long int	32	-2147483648 à 21474483647	4
long double	64	-1.7E-308 à 1.7E+308	8

5. Operadores

São símbolos especiais que obrigam o compilador a executar determinadas operações. Estas operações podem ser aritméticas, comparativas ou lógicas.

5.1 - Operadores aritméticos

São operadores que realizam uma operação matemática.

Tabela 5: Operadores aritméticos

Operador aritmético	Ação
-	Subtração
+	Adição
*	Multiplicação
/	Divisão
%	Resto inteiro da divisão
-- ++	Decremento/incremento

Precedência dos operadores aritméticos (Hierarquia nas Operações)

Tabela 6: Precedência dos operadores aritméticos

Hierarquia	Operação
1	Parênteses
2	Funções
3	++ --
4	- (menos unário)
5	* / %
6	+ -

Observação: Quando houver duas ou mais operações de mesma hierarquia, o compilador executa-as da **esquerda** para a **direita**.

5.2 - Operadores relacionais

São operadores que permitem comparar valores, ou seja, são utilizados principalmemte em comandos que possuem **condições**.

Tabela 7: Operadores relacionais

Operador	Ação
>	Maior que
>=	Maior ou igual a
<	Menor que
<=	Menor ou igual
==	Igual a
!=	Diferente de

5.3 - Operadores lógicos

São operadores utilizados em comandos que tem mais de uma **condição**.

Exemplo: if (condição1 **&&** condição2 **||** condição3)

Tabela 8: Operadores lógicos

Operador lógica	Ação
&&	AND (e)
||	OR (ou)
!	NOT (não)

Precedência (Hierarquia dos operadores relacionais e lógicos)

Tabela 9: Precedência dos operadores relacionais e lógicos

Hierarquia	Operação
1	!
2	> >= < <=
3	== !=
4	&&
5	||

Observação: As expressões que utilizam operadores relacionais e lógicos retornam **0** (zero) para falso e **!0** (não zero) para verdadeiro, ou seja:

```
#define TRUE   !0
#define FALSE  0
```

5.4 - Incremento e decremento

São operadores aritméticos que permitem realizar operações de soma e subtração de forma simplificada.

- ++ adiciona (1) ao operando
- subtrai (1) ao operando

As seguintes operações são equivalentes:

```
x++;          x = x + 1;
x--;          x = x - 1;
```

Observação:

Os operadores (++ ou --) podem ser colocados antes ou depois do operando. Quando precede seu operando, C efetua a **operação** de incremento ou decremento antes de utilizar o valor do operando. Quando o operador vier depois do operando, C utiliza o **valor do operando** antes de incrementá-lo ou decrementá-lo.

Exemplo:

```
x = 10;                 /*  y será 11  */
y = ++x;        /*  x será 11  */

x = 10;                 /*   y será 10  */
y = x++;        /*  x será 11  */
```

5.5 - Operador de atribuição

O operador de atribuição é o sinal de igual =. A sintaxe do operador de atribuição pode ser escrito em uma das seguintes formas:

```
variável = constante;          x = 3;
variável = variável;           x = y;
variável = expressão;          x = a + b;
variável = função(x);          x = sqrt(y);
```

Programa exemplo (3): O programa calcula a idade de uma pessoa.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int idade, ano_atual, ano_nasceu;

 clrscr();
 printf("Ano ATUAL: ");
```

```
 scanf("%d",&ano_atual);                    /* leitura do ano atual */
 printf("Ano em que NASCEU: ");
 scanf("%d",&ano_nasceu);                   /* leitura do ano de nascimento */
 idade = ano_atual - ano_nasceu;            /* atribuição – cálculo da idade */
 printf("Sua IDADE e\' %d\n",idade);
 getch();
}
```

A linguagem de programação C permite utilizar o operador de atribuição em expressões, junto com operadores matemáticos, lógicos, relacionais, chamada de funções, e outros (como foi mencionado acima).

```
if ((produto = x * y) < 0)
```

Funcionamento: Primeiramente C atribui o valor x * y a variável **produto**, para depois avaliar a expressão, ou seja, comparar se o **produto** é menor (<) que zero.

5.6 - O operador sizeof

O operador **sizeof** (tamanho de) retorna o tamanho (em bytes) de uma variável ou de um tipo que está em seu operando.

Programa exemplo (4): O programa exibe a quantidade de bytes das variáveis e tipos.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int x, y;

 clrscr();
 x = sizeof(float);                         /* x vale 4 */
 printf("%d\n%d",x,sizeof(y));              /* sizeof(y) é 2 */
 getch();
}
```

5.7 - Casts

É possível forçar que o resultado de uma expressão seja de um determinado tipo. Para tanto deve ser utilizado uma construção chamada de **cast**, ou seja, pode ser utilizado para "**tipar**" uma variável com um tipo diferente do resultado da expressão.

```
variável = (tipo) expressão;
```

Programa exemplo (5): O programa imprime na tela o resultado de uma divisão.

```
#include <stdio.h>
#include <conio.h>
```

```
void main (void)
{
 int x, y;
 float resp;

 clrscr;
 printf("x = ");
 scanf("%d",&x);
 printf("y = ");
 scanf("%d",&y);
 resp = (float) x / y;                  /* é necessário um cast (float) pois a divisão de dois */
 printf("Divisao = %.2f\n",resp);       /* inteiros resulta em um inteiro */
 getch();
}
```

Observação: Em C, o tipo resultante de um inteiro dividido por outro inteiro é um inteiro, logo, deve-se utilizar um **cast** (float) para que o tipo resultante atribuído a variável **resp** seja **float**.

5.8 - Expressões

Uma expressão em C é qualquer combinação válida de **operadores** (aritméticos, relacionais ou lógicos), **constantes**, **funções** e **variáveis**.

Exemplo: c = sqrt (a) + b / 3.4;

5.8.1 - Conversão de tipos em expressões

Quando uma expressão é composta de tipos diferentes, C converte todos os operandos para o tipo do maior operando. Antes, os tipos abaixo são convertidos para:

char - convertido para **int**
float - convertido para **double**

Exemplo:

```
char ch;
int i;
float f;
double d;
float result;
```

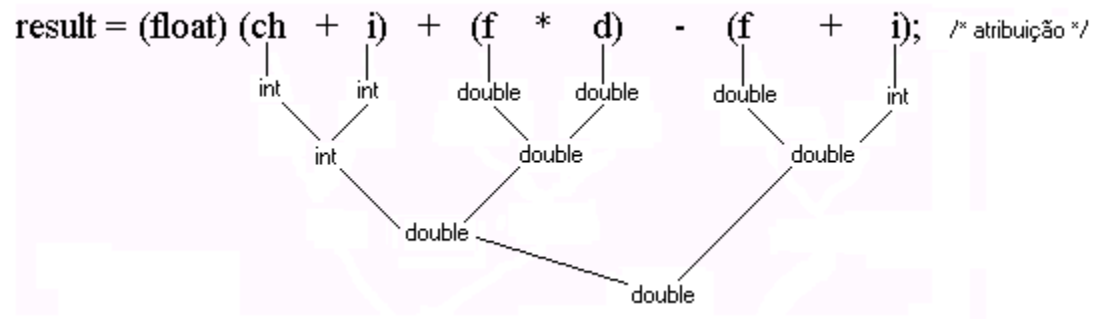



Figura 3: Conversão de tipos em expressões

6. Funções padrões

6.1 - abs

A função **abs** retorna o valor inteiro positivo – **absoluto**.

Sintaxe: int **abs** (int x);
Prototype: math.h e stdlib.h

6.2 – fabs

A função **fabs** retorna o valor real positivo – **absoluto**.

Sintaxe: float **fabs** (float x);
Prototype: math.h e stdlib.h

6.3 – asin

A função **asin** retorna o valor do **arco seno**. A variável **x** deve estar em radianos.

Sintaxe: double **asin** (double x);
Prototype: math.h
Faixa: -pi / 2 à pi / 2

6.4 – acos

A função **acos** retorna o valor do **arco cosseno**. A variável **x** deve estar em radianos.

Sintaxe: double **acos** (double x);
Prototype: math.h
Faixa: 0 à pi

6.5 – atan

A função **atan** retorna o valor do **arco tangente**. A variável **x** deve estar em radianos.

Sintaxe: double **atan** (double x);
Prototype: math.h
Faixa: -pi / 2 à pi / 2

6.6 – cos

A função **cos** retorna o valor do **cosseno**. A variável **x** deve estar em radianos.

Sintaxe: double **cos** (double x);
Prototype: math.h
Faixa: -1 à 1

6.7 – sin

A função **sin** retorna o valor do **seno**. A variável **x** deve estar em radianos.

Sintaxe: double **sin** (double x);
Prototype: math.h
Faixa: -1 à 1

6.8 – exp

A função **exp** retorna o valor do **expoente** (**e**x).

Sintaxe: double **exp** (double x);
Prototype: math.h

6.9 – pow

A função **pow** (**pow**er) retorna o valor da **potência** (**x**y).

Sintaxe: double **pow** (double x, double y);
Prototype: math.h

6.10 – sqrt

A função **sqrt** (**sq**uare **r**oo**t**) retorna o valor da **raiz quadrada**.

Sintaxe: double **sqrt** (double x);
Prototype: math.h

6.11 – log

A função **log** retorna o valor do **logaritmo natural**.

Sintaxe: double **log** (double x);
Prototype: math.h

6.12 – atof

A função **atof** converte **string** em **ponto flutuante**.

Sintaxe: double **atof** (const char *s);
Prototype: math.h e stdlib.h

6.13 – atoi

A função **atoi** converte uma **string** em **inteiro**.

Sintaxe: int **atoi** (const char *s);
Prototype: stdlib.h

6.14 – atol

A função **atol** converte uma **string** em **inteiro longo**.

Sintaxe: long int **atol** (const char *s);
Prototype: stdlib.h

6.15 - log10

A função **log10** retorna o **logarítmo na base 10**.

Sintaxe: double **log10** (double x);
Prototype: math.h

6.16 – tan

A função **tan** retorna o valor da **tangente**. A variável **x** deve estar em radianos.

Sintaxe: double **tan** (double x);
Prototype: math.h

6.17 – max

A função **max** retorna o **maior** valor entre dois valores.

Sintaxe: int **max** (int a, int b);
Prototype: stdlib.h

6.18 – min

A função **min** retorna o **menor** valor entre dois valores.

Sintaxe: int **min** (int a, int b);
Prototype: stdlib.h

6.19 – rand

A função **rand** retorna um **número aleatário** entre 0 até 32767.

Sintaxe: int **rand** (void);
Prototype: stdlib.h
Faixa: 0 à 32767

6.20 – random

A função **random** retorna um **número aleatório** entre 0 até (limite – 1).

Sintaxe: int **random** (int limite);
Prototype: stdlib.h
Faixa: 0 à (limite - 1)

6.21 – randomize

A função **randomize** inicializa a geração de números aleatórios.

Sintaxe: void randomize (void);
Prototype: stdlib.h

6.22 – system

A função **system** executa comandos e arquivos .COM e .EXE do sistema operacional.

Sintaxe: int **system** (const char *comando);
Prototype: process.h e stdlib.h
Retorna: 0 (ok) e -1 (erro)

Exemplos:

```
system ("dir");
system ("sk");
system ("dir *.c");
```

7. Comandos

7.1 - Tipos de Comandos

7.1.1 - Seqüência

São comandos, que no fluxo de controle do programa, são sempre executados passando a execução para a próxima instrução, ou seja, todos os comandos de seqüência são executados desde que eles não dependem de um comando de seleção.

Exemplo: (todas as instruções abaixo são de seqüência)

```
clrscr();                          /* limpa a tela */
printf("Digite uma letra: ");      /* imprime na tela */
letra = getch();                   /* atribuição */
printf("Digite um valor: ");
scanf("%f",&Valor);                /* entrada de dados via teclado */
resp = valor * 1.25;               /* atribuição é um comando de seqüência */
```

Observação: A função **getche**() permite a entrada de um caracter via teclado. Não é necessário o usuário digitar <**enter**>.

7.1.2 - Seleção

São comandos, que no fluxo de controle do programa, permitem a seleção entre duas ou mais instruções, ou seja, este tipo de comando faz com que alguns comandos não sejam executados.

Exemplo:

```
if (numero % 2 == 0)          /* testa se o número é par ou ímpar */
    printf("Número: PAR\n");
else
    printf("Número: ÍMPAR\n");
```

ou

```
#include <stdio.h>
#include <conio.h>
#include <math.h>             /* por causa da função fabs */
#include <stdlib.h>           /* por causa da função fabs */

void main (void)
{
  float x, raiz;

  clrscr();
```

```
  printf("Digite um valor: ");
  scanf("%f", &x);
  if (x < 0)
     x = fabs(x);                    /* esta instrução só é executada se o valor de x for negativo */
  raiz = sqrt(x);
  printf("Raiz Quadrada: %.2f\n", raiz);
  getche();
}
```

7.1.3 - Repetição

São comandos, que no fluxo de controle do programa, permitem a repetição de uma ou mais instruções.

Programa Exemplo (6): O programa exibe na tela a tecla e o código (ASC II) da tecla digitada pelo usuário. O programa encerra quando o usuário digitar a tecla <**esc**> (**escape**).

```
#include <stdio.h>
#include <conio.h>

#define ESC     27

void main (void)
{
 char tecla;

 clrscr();
 do {
       tecla = getch();
       printf("Tecla: %c →  Código: %d\n", tecla, tecla);
      } while (tecla != ESC);
}
```

7.1.4 Atribuição

Veja ítem 5.5 (Operador de atribuição).

7.2 - Comando if

O comando **if** é um comando de seleção que permite selecionar um comando entre dois outros comandos (comandos simples) ou dois conjuntos de comandos (comandos compostos). Isto é feito através da avaliação de uma condição. O resultado desta avaliação (teste da condição) pode ser **verdadeiro** ou **falso**. Dependendo deste resultado um dos comandos é executado e o outro não.

Sintaxe:

```
if (condição)                                              if (condição)
     comando 1;                ou                     comando;
else
     comando 2;          /* todos os comando são simples */
```

Observação: O **else** é opcional.

Se a condição for avaliada como verdadeira (qualquer valor diferente de 0), o comando 1 será executado, caso contrário (condição falsa, valor igual a zero) o comando 2 será executado. Comando 1, comando 2 ou comando podem ser **simples** ou **compostos** (quando há mais de um comando ligado a outro, deve-se utilizar chaves ({ }). Veja exemplos abaixo

```
if (condição)
  {
    comando 1;
    comando 2;
  }
else
      {
       comando 3;
       comando 4;
     }
```

```
if (condição)
  {
   comando 1;
   comando 2;
   comando 3;
  }

/* todos os comando são compostos */
```

Programa exemplo (7): O usuário digita um número e o programa diz se este é **par** ou **ímpar**.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int numero;

 clrscr();
 printf("Digite um número: ");
 scanf("%d", &numero);
 if ((numero % 2) == 0)
      printf("Número é PAR\n");              /* comando simples */
 else
      printf("Número é IMPAR\n");                        /* comando simples */
 getch();
}
```

7.2.1 - if encadeados

Um **if** aninhado (ou encadeado) é um comando **if** dentro de outro comando **if** ou **if** ... **else**.

Programa exemplo (8): O usuário digita um número e o programa diz se este é **zero**, **positivo** ou **negativo**.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int num;
```

```
 clrscr();
 printf("Digite um número: ");
 scanf("%d", &num);
 if (num == 0)
     printf("Zero\n");                  /* comando 1 do primeiro if */
 else
     if (num > 0)
         printf("Positivo\n");
     else                               /* comando 2 do primeiro if */
         printf("Negativo\n");
 getch();
}
```

Observação: Note que no comando 2 (do primeiro **if**) existe outro **if.**

Exemplo:

```
if (x)                 /* primeiro if */
  if (y)               /* segundo if */
      printf("1");
  else
     printf("2");
```

<u>**Primeira dúvida**</u>: O **else** pertence a qual **if**?

Funcionamento: Em **C** o **else** está ligado ao **if** mais próximo (mais interno), ou seja, neste exemplo o **else** pertence ao segundo **if.**

<u>**Segunda dúvida**</u>: Como fazer para que o **else** (do exemplo acima) pertença ao primeiro **if**?

Resposta: Deve-se utilizar chaves para anular a associação normal (veja exemplo abaixo).

Exemplo:

```
if (x)
  {
   if (y)
     printf("1");
  }
else
    printf("2");
```

7.3 - O comando switch

O comando **switch** é um comando de seleção que permite selecionar um comando entre vários outros comandos. Isto é feito através da comparação de

uma variável a um conjunto de constantes. Cada um dos comandos está ligado a uma constante.

Sintaxe:

```
switch (variável)
        {
          case constante_1 :  seqüência de comandos;
                              break;
          case constante_2 :  seqüência de comandos;
                              break;
                 .
                 .
                 .
          default: seqüência de comandos;
        }
```

O programa testa uma variável sucessivamente contra uma lista de constantes inteiras ou caracteres (**int** ou **char**). Depois de encontrar uma coincidência, o programa executa o comando ou bloco de comandos que estejam associados àquela constante. O comando **default** é executado se não houver nenhuma coincidência.

O comando **break** é utilizado para obrigar a saída do comando **switch**.
A opção **default** é opcional.

Observação: A variável não pode ser uma **string** (char *) e nem **real** (float).

Programa exemplo (9): O programa recebe um dígito de 0 à 9 e imprime na tela, este dígito, por extenso. Neste exemplo a variável dígito é inteira.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int digito;

 clrscr();
 printf("Dígito [0 .. 9]: ");
 scanf("%d", &digito);
 switch (digito)
       {
        case 0: printf("Zero\n");
                  break;
        case 1: printf("Um\n");
                  break;
        case 2: printf("Dois\n");
                  break;
        case 3: printf("Três\n");
                  break;
        case 4: printf("Quatro\n");
                  break;
        case 5: printf("Cinco\n");
                  break;
        case 6: printf("Seis\n");
```

```
            break;
     case 7: printf("Sete\n");
            break;
     case 8: printf("Oito\n");
            break;
     case 9: printf("Nove\n");
            break;
    default: printf("ERRO: Não é um digito\n");
   }
 getch();
}
```

Programa exemplo (10): O programa recebe um dígito de 0 à 9 e imprime na tela, este dígito, por extenso. Neste exemplo a variável dígito é caracter, por causa disto as constantes estão entre apostrofes.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 char digito;

 clrscr();
 printf("Dígito [0 .. 9]: ");
 scanf("%c", &digito);
 switch (digito)
    {
     case '0': printf("Zero\n");
              break;
     case '1': printf("Um\n");
              break;
     case '2': printf("Dois\n");
              break;
     case ‘3’: printf("Três\n");
              break;
     case ‘4’: printf("Quatro\n");
              break;
     case ‘5’: printf("Cinco\n");
              break;
     case ‘6’: printf("Seis\n");
              break;
     case ‘7’: printf("Sete\n");
              break;
     case ‘8’: printf("Oito\n");
              break;
     case '9': printf("Nove\n");
              break;
    default: printf("ERRO: Não é um digito\n");
   }
 getch();
}
```

Programa exemplo (11): O programa constrói um menu com três funções: **inclusão**, **alteração** e **término**.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 char opcao;

 clrscr();
 printf("[I]nclusão\n");
 printf("[A]lteração\n");
 printf("[T]érmino\n");
 printf("Qual a opção: ");
 opcao = getche();
 switch (opcao)
       {
        case 'i':
        case 'I': inclusao();
                  break;
        case 'a':
        case 'A': alteracao();
                  break;
        case 't':
        case 'T': termino();
                  break;
       default: printf("ERRO: Opção Inválida\n");
      }
 getch();
}

void inclusao()
{
}

void alteracao()
{
}

void termino()
{
}
```

7.4 - Comando while

O comando **while** é um comando de repetição que permite executar um comando (simples) ou vários comandos (composto) diversas vezes. Isto é feito através da avaliação de uma condição. Enquanto a condição for verdadeira os comandos são repetidos. Quando a condição se tornar falsa o comando **while** é finalizado. O teste da condição é feita no início do comando, ou seja, antes que todos os comandos sejam executados.

Observação: Note que os comandos podem não ser executados nenhuma vez, basta a condição começar como falsa.

Sintaxe:

```
while (condição)                  ou      while (condição)
        comando;                              {
                                              comando 1;
                                              comando 2;
/* comando simples */                           }

                                          /* comando composto */
```

Condição: Qualquer expressão válida em C com resultado 0 (**false**) ou !0 (**true**). Na condição podem ser utilizados ainda **variáveis**, **constantes**, **funções**, **operadores** (aritméticos, relacionais e lógicos).

Funcionamento do comando: O loop (laço) é repetido enquanto a condição for verdadeira. Quando a condição se tornar falsa o controle do programa passa para a próxima instrução. O loop **while** verifica a condição no início do laço, por causa disto, normalmente, a variável de controle do laço deve ser inicializado.

Exemplo:

```
int i = 0;          /* inicialização da variável de controle */
while (i <= 10)             /* condição i <= 10 */
        {
          printf("i = %d\n", i);
          i = i + 1;          /* incremento */
        }
```

Comando: Pode ser um comando vazio, simples ou composto que serão repetidos.

Comando vazio:

```
while (!kbhit());               /* comando while não repete nenhum comando */
for (i = 0; i <= 1000; i++);  /* comando for não repete nenhum comando */
```

Verifique: Note que no final dos dois comandos (**while** e **for**) existe apenas um ponto-e-vírgula, isto é o sinal de comando vazio, ou seja, os comandos **while** e **for** que teriam outros comandos não os tem, caracterizando comandos vazios.

Problema freqüente de digitação: Muitas vezes o programador insere um ponto-e-vírgula no final de um comando **for** ou **while** por engano. Isto é um grave problema, pois este ponto-e-vírgula (inserido acidentalmente) faz com que os comandos que seriam repetidos, não são. Veja o exemplo abaixo:

Exemplo:

```
for (x = 1; x <= 10; x++);     /* note o ponto-e-vírgula no final do comando for */
printf("x = %d\n", x);        /* é impresso x = 11 na tela, porque? */
```

Explicação: O comando **printf** não faz parte do comando **if** devido ao ponto-e-vírgula no comando **for**. O comando **for** termina quando a variável de controle **x** chega ao valor 11.

Comando correto: **for** (x = 1; x <= 10; x++) **printf**(“x = %d\n”, x)**;**

Programa exemplo (12): O programa imprime caracteres de ‘A’ até ‘Z’ na tela.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 char letra = ‘A’;                              /* inicialização da variável de controle */

 clrscr();
 while (letra != ‘Z’)
      {
       printf ("Letra: %c\n", letra);
       letra++;                                 /* incremento */
      }
 getch();
}
```

7.5 - O comando for

O comando **for** é um comando de repetição que permite executar um comando (comando simples) ou vários comandos (comando composto) diversas vezes. Isto é feito através da avaliação de uma condição. Enquanto a condição for verdadeira os comandos são repetidos. Quando a condição se tornar falsa o comando **for** é finalizado.

Sintaxe: **for** (inicialização; condição; incremento ou decremento)
comando;

Inicialização: É um comando de atribuição (ou vários, separados por vírgula) que o compilador utiliza para inicializar a(s) variável(is) de controle do laço.

Condição: É uma expressão qualquer, que testa a variável de controle do laço contra algum valor para determinar quando o laço terminará.

Incremento ou decremento: Define a maneira como a(s) variável(is) de controle do laço serão alteradas após a repetição do laço.

- O laço (**for**) é repetido enquanto a condição é verdadeira.
- A condição é sempre testada no começo do laço.
- Qualquer uma das 3 partes do comando **for** (inicialização; condição; incremento) podem ser qualquer expressão válida em C.

Programa exemplo (13): O programa imprime de 1 até 10 na tela.

```
#include <stdio.h>
```

```
#include <conio.h>

void main (void)
{
 int i;

 clrscr();
 for (i = 1; i <= 10; i++)              /* inicialização: i = 1 */
    printf("%d\n",i);                   /* condição: i <= 10 */
 getch();                               /* incremento: i++ */
}
```

Programa exemplo (14): O programa imprime na tela:

```
i = 1 → j = 9
i = 2 → j = 8
i = 3 → j = 7
i = 4 → j = 6
```

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int i, j;

 clrscr();
 for (i = 1,,j = 9; i != j; i++,,j--)
     printf("i = %d → j = %d\n", i ,j);
 getch();
}
```

O laço **for** é equivalente ao seguinte comando:

```
inicialização;
while (condição)
     {
      comando;
      incremento;           /* ou decremento */
     }
```

ou

```
inicialização;
do {
      comando;
      incremento;           /* ou decremento */
    } while (condição);
```

7.6 - O loop do { } while

O comando **do** ... **while** é um comando de repetição que permite executar um comando (comando simples) ou vários comandos (comando composto) diversas vezes. Isto é feito através do teste de uma condição. Enquanto a condição for verdadeira os comandos são repetidos. Quando a condição se tornar falsa o comando **do** ... **while** é finalizado. O teste da condição é feita no final do

comando, ou seja, depois que os comandos são executados. (Note que os comandos são executados pelo menos uma vez).

Sintaxe:

```
do {
      comando;
    } while (condição);
```

- Repete o laço enquanto a condição for verdadeira.
- Testa a condição no final, fazendo com que o laço seja executado pelo menos uma vez.

Programa exemplo (15): Imprime na tela de 1 até 10.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 int i = 1;

 clrscr();
 do {
      printf("%d\n",i);
      i++;
    } while (i <= 10);
}
```

7.7 - Interrupção de loops

7.7.1 - O comando break

Quando o programa encontra o comando **break** dentro de um laço, ele imediatamente encerra o laço, e o controle do programa segue para o próximo comando após o laço.

Programa exemplo (16): O programa imprime na tela a tecla digitada pelo usuário até que ele digite <***esc***>.

```
#include <stdio.h>
#include <conio.h>

#define ESC     27

void main(void)
{
 char tecla;

 clrscr();
 do {
      tecla = getche();
      if (tecla == ESC)                    /* encerra o laço quando o usuário teclar ESC */
        break;
      printf("Tecla: %c\n", tecla);
```

```
    } while (1);                    /* laço eterno */
}
```

7.7.2 - O comando continue

O comando **continue** em vez de forçar o encerramento, força a próxima interação do laço e "pula", ou seja, não executa o código que estiver depois do comando **continue.**

Programa exemplo (17): O programa imprime na tela somente os números pares de 0 até 100.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 int i;

 clrscr();
 for (i = 0; i < 100; i++)
     {
      if (i % 2)                    /* 0 é par, 1 é impar */
         continue;
      printf("Par: %d\n", i);                /* imprime somente números pares */
     }
}
```

Nos laços **while** e **do {} while** um comando **continue** faz com que o controle do programa execute diretamente o teste da condição e depois continue o processo do laço.

No caso do comando **for**, o programa primeiro executa o incremento (ou decremento) do laço e, depois, executa o teste da condição antes de finalmente fazer o laço continuar.

7.8 - A função exit ()

A função **exit** aborta o programa em qualquer situação.

Modo de usar: **exit**(0); ou **exit**(!0);

8. Entrada e saída

Em C, as entradas e saídas são realizadas através da utilização das funções da biblioteca padrão do C, algumas são encontradas no arquivo **stdio.h**.

8.1 - Entrada e saída do console

As seguintes funções estão definidas em **stdio.h** e **conio.h**.

Tabela 10: Funções de entrada e saída via console

Função	**Efeito** (Entrada)
getchar()	Lê um caracter do teclado; espera por <**enter**>
getche()	Lê um caracter e imprime na tela; não espera por <**enter**>
getch()	Lê um caracter sem imprimir na tela; não espera por <**enter**>
putchar()	Escreve um caracter na tela
gets()	Lê uma *string* do teclado (aceita espaço)
puts()	Escreve uma *string* na tela

8.2 - Entrada e saída formatada

8.2.1 - Saída formatada (printf)

Sintaxe: **printf** ("string de controle", lista de variáveis);

String de controle: Formada pelos caracteres que a função imprime na tela, e pelos comandos de formatação (**%c**, **%s**, **%d**, **%f**) que definem a maneira como as variáveis serão impressas e caracteres especiais (**\n**, **\t**, ...).

Tabela 11: Comandos de formatação

Código	**Tipo**	**Formato**
%s	char *	String (vetor de caracteres)
%d	int	Inteiro decimal com sinal
%i	int	Inteiro decimal com sinal
%o	int	Inteiro octal sem sinal
%u	int	Inteiro decimal sem sinal
%x	int	Inteiro hexadecimal sem sinal (com a, b, c, d, e, f)
%X	int	Inteiro hexadecimal sem sinal (com A, B, C, D, E, F)
%f	float	Valor com sinal da forma [-]dddd.dddd
%e	float	Valor com sinal da forma [-]d.dddd e [+/-]ddd
%g	float	Valor com sinal na forma **e** ou **f** baseado na precisão do valor dado
%E	float	Mesmo que **e**, mas com E para expoente
%G	float	Mesmo que **g**, mas com E para expoente
%c	char	Um caracter
%%	nada	O caracter % é impresso
%n	int *	Armazena o número de caracteres escritos até o momento
%p	ponteiro	imprime como um ponteiro

Flags (Bandeiras):

- (-) Alinha o resultado à esquerda. Preenche o restante do campo com brancos. Se não é colocado, alinha o resultado à direita e preenche o restante à esquerda com zeros ou brancos.
- (+) O resultado sempre começa com o sinal + ou -
- (#) Especifica que o argumento será impresso usando uma das formas alternativas

Formas alternativas:

- 0 É colocado zeros (0) antes do argumento
- x ou X É colocado 0x (ou 0X) antes do argumento

Especificadores de largura do campo a ser impresso (exemplos):

Tabela 12: Especificadores de largura do campo

Prefixo	**6d**	**6o**	**8x**	**10.2e**	**10.2f**
%-+#0	+00555	01053	0x0022b	+5.50e+000	+000005.50
%-+#	+555	01053	0x22b	+5.50e+000	+5.50
%-+0	+00555	01053	000022b	+5.50e+000	+000005.50
%-+	+555	1053	22b	+5.50e+000	+5.50
%-#0	000555	001053	0x00022b	05.50e+000	0000005.50
%-#	555	01053	0x22b	5.50e+000	5.50
%-0	000555	01053	0000022b	05.50e+000	0000005.50
%-	555	1053	22b	5.50e+000	5.50
%+#0	+00555	01053	0x0022b	+5.50e+000	+000005.50
%+#	+555	01053	0x22b	+5.50e+000	+5.50
%+0	+00555	01053	000022b	+5.50e+000	+000005.50
%+	+555	1053	22b	+5.50e+000	+5.50
%#0	000555	001053	0x00022b	05.50e+000	0000005.50
%#	555	01053	0x22b	5.50e+000	5.50
%0	000555	001053	0000022b	05.50e+000	0000005.50
%	555	1053	22b	5.50e+000	5.50

8.2.2 - Entrada formatada (scanf)

Sintaxe: **scanf** ("string de controle", lista de variáveis);

String de controle: Local onde é definido o tipo de dado (**%d**, **%c**, **%s**, **&f**, ...) que será lido pelo teclado (não deve conter mais nenhum caracter).

Lista de variáveis: Nome da(s) variável(is) que será(ão) lida(s) pelo teclado.

Observação: Deve ser passado o endereço do argumento a ser lido.

```
int x;

scanf("%d",&x);
```

Programa exemplo (18): O programa permite a entrada (via teclado) do **nome**, **idade** e **salário** de uma pessoa.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
```

```
 int idade;
 float salário;
 char nome[40];

 clrscr();
 printf("Qual seu nome: ");
 scanf("%s", nome);            /* ATENÇÃO: nome é igual ao &nome[0] */
 printf("Idade: ");
 scanf("%d",&idade);           /* &idade é o endereço da variável idade */
 printf("Salário: ");
 scanf("%d",&salário);         /* &salário é o endereço da variável salário */
 getch();
}
```

8.3 - Saída na impressora (fprintf)

Sintaxe: **fprintf** (stdprn, "string de controle", lista de variáveis);

stdprn: Fila de impressão (**st**an**d**ard **pr**i**n**ter)

Programa exemplo (19): O programa imprime "**UCPel**" na impressora.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 clrscr();
 fprintf(stdprn,"UCPel – Universidade Católica de Pelotas\n");
}
```

Observação:

```
/* -------------------------- Salta uma página na impressora */
 fprintf(stdprn,"%c\n",12);
/* -------------------- Comprime os caracteres na impressora */
 fprintf(stdprn,"%c\n",15);
```

9. Controle do vídeo e teclado

9.1 – clrscr

A função **clrscr** (**cl**ea**r** **scr**een) pré-definida do C que permite limpar toda tela, o cursor permanece no canto superior esquerdo.

Sintaxe: void **clrscr**(void);
Prototype: conio.h

9.2 – gotoxy

A função **gotoxy** (coluna, linha) (vá para posição x, y) pré-definida do C que permite o posicionamento do cursor em qualquer posição da tela.

Sintaxe: void **gotoxy** (int coluna, int linha);
Prototype: conio.h

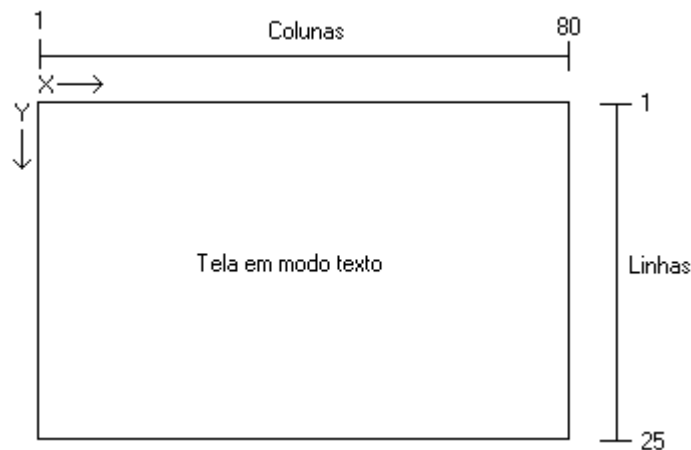



Figura 4: Tela em modo texto

Programa exemplo (20): O programa imprime o título "**calculadora**" centralizado na primeira linha da tela.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>                    /* devido a função strlen */

void main(void)
{
 int col, lin = 1, n;
 char titulo[] = "Calculadora";      /* a string título é definida com 11 caracteres (NULL) */

 clrscr();
 n = strlen(titulo);            /* strlen(tirulo) é igual a 10, ou seja, quantidade de caracteres da string */
 col = (80 - n) / 2;            /* cálculo da centralização do título na linha */
 gotoxy(col, lin);
 printf("%s", titulo);
 getch();
}
```

9.3 – clreol

A função **clreol** (**cl**ea**r** **e**nd **o**f **l**ine) pré-definida do C que permite que uma linha seja apagada desde a posição do cursor p (x, y) até o final da linha. Deve ser utilizado a função **gotoxy** (c, l) para posicionar o cursor.

Sintaxe: void **clreol**(void);

Prototype: conio.h

9.4 – delline

A função **delline** (**del**ete **line**) pré-definida do C que permite que uma linha seja apagada. Deve ser utilizado a função **gotoxy** (c, l) antes.

Sintaxe: void **delline**(void);
Prototype: conio.h

10. Comandos especiais

10.1 – delay

A função **delay** (pausa) pré-definida do C que permite uma parada temporária (em milisegundos) na execução de um programa.

Sintaxe: void **delay**(unsigned int x); /* x em milisegundos */
Prototype: dos.h

10.2 – sleep

A função **sleep** (pausa) pré-definida do C que permite uma parada temporária (em segundos) na execução de um programa.

Sintaxe: void **sleep**(unsigned int x); /* x em segundos */
Prototype: dos.h

10.3 textbackground

A função **textbackground** (altera a cor de fundo) pré-definida do C que permite a mudança na cor de fundo da tela. Podem ser utilizadas 8 cores.

Sintaxe: void **textbackground** (int cor);
Prototype: conio.h

Tabela 13: Cores de fundo

Número	Nome	Cor
0	BLACK	Preto
1	BLUE	Azul
2	GREEN	Verde
3	CYAN	Azul claro
4	RED	Vermelho
5	MAGENTA	Rosa

6	BROWN	Marrom
7	LIGHTGRAY	Cinza claro

<u>**Observação**</u>: Para a cor pode ser utilizado o número da cor ou o nome em letras **maiúsculas**.

10.4 – textcolor

A função **textcolor** (altera a cor de frente, ou seja, a cor do texto)pré-definida do C que permite mudança na cor de texto exibido na tela. É possível utilizar 16 cores.

Sintaxe: void **textcolor**(int cor); /* cor: número ou nome da cor */
Prototype: conio.h

Tabela 14: Cores da frente (texto)

Número	Identificador	Cor
0	BLACK	Preto
1	BLUE	Azul
2	GREEN	Verde
3	CYAN	Azul claro
4	RED	Vermelho
5	MAGENTA	Rosa
6	BROWN	Marrom
7	LIGHTGRAY	Cinza claro
8	DARKGRAY	Cinza escuro
9	LIGHTBLUE	Azul claro
10	LIGHTGREEN	Verde claro
11	LIGYTCYAN	Azul claro
12	LIGHTRED	Vermelho claro
13	LIGHTMAGENTA	Rosa claro
14	YELLOW	Amarelo
15	WHITE	Branco
128	BLINK	Piscante

10.5 – window

A função **window** (define uma janela ativa) pré-definida do C que permite uma mudança na janela de texto ativa, ou seja, parte da tela onde caracteres podem ser impressos. Normalmente a janela ativa é toda tela, ou seja:

window(1, 1, 80, 25);

Sintaxe: void **window** (unsigned int left, unsigned int top, unsigned int right, unsigned int bottom);
Prototype: conio.h

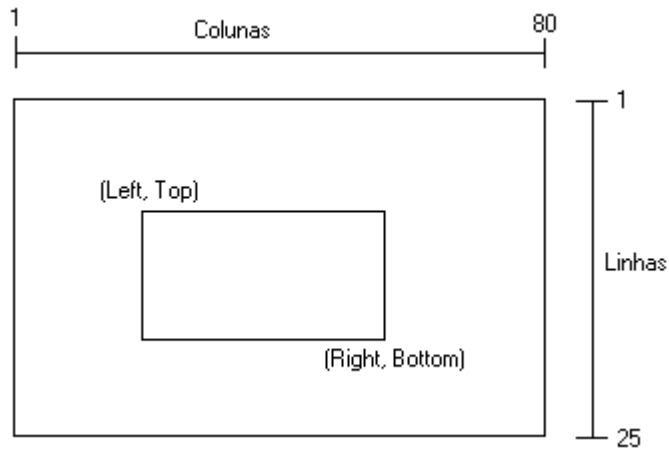



Figura 5: Coordenadas de tela em modo texto

Observação: Após selecionar uma janela ativa a posição (1,1) passa a ser o canto superior esquerdo da janela selecionada.

10.6 – sound e nosound

sound: Função pré-definida do C que ativa o auto-falante do computador.

Sintaxe: void **sound**(unsigned int freqüência);
Prototype: dos.h

nosound: Função pré-definida do C que desativa o auto-falante do computador.

Sintaxe: void **nosound**(void);
Prototype: dos.h

Programa exemplo (21): O programa ativa (e desativa) o auto-falante do computador.

```
#include <conio.h>
#include <dos.h>

void main(void)
{
 sound(220);                /* ativa o auto-falante com a freqüência 220 */
 delay(200);                /* período de duração do som - pausa*/
 nosound();                 /* desativa o auto-falante */
}
```

10.7 – wherex e wherey

A funções **wherex** e **wherey** (informa a posição do cursor) que indicam a posição corrente do cursor.

Sintaxe: int **wherex**(void); e int **wherey**(void);
Prototype: conio.h

Programa exemplo (22): O programa exibe a posição do *mouse* (coluna e linha).

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 int coluna, linha;

 coluna = wherex();
 linha = wherey();
 printf("Coluna: %d\n", coluna);
 printf("Linha : %d\n", linha);
 getch();
}
```

10.8 – textmode

A função **textmode** (permite alterar o modo de texto) pré-definida do C que permite redefinir o modo do texto na tela, ou seja, colorido ou preto-e-branco, 80 ou 40 colunas.

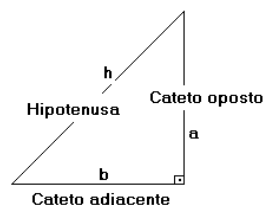
Sintaxe: void **textmode** (int modo);
Prototype: conio.h

Tabela 15: Modos de texto

Modos	**Resolução da tela** (colunas x linhas)
C80	Colorido (80 x 25)
C40	Colorido (40 x 25)
BW80	Preto e branco (80 x 25)
BW40	Preto e branco (40 x 25)

10.9 – Lista de exercícios (comandos)

✓ Escreva um programa em C que recebe dois valores via teclado: **cateto adjacente** (b) e **cateto oposto** (a) e calcula o valor da **hipotenusa** dado pela seguinte fórmula:

Fórmula: $h^2 = a^2 + b^2$

Exemplo (Tela):

Cateto adjacente (b): **3** <**enter**>
Cateto oposto (a): **4** <**enter**>
Hipotenusa: **5**

✓ Escreva um programa em C que lê 4 notas via teclado: **n1**, **n2**, **n3** e **n4** obtidas por um aluno em 4 avaliações. Calcule a **média** utilizando a seguinte fórmula:

$$\text{Média} = \frac{n1 + n2 * 2 + n3 * 3 + n4}{7}$$

A seguir imprima na tela a **média** e o **conceito** do aluno baseado na seguinte tabela:

Média	**Conceito**
9,0 ou acima de 9,0	A
entre 7,5 (inclusive) e 9,0	B
entre 6,0 (inclusive) e 7,5	C
abaixo de 6,0	D

✓ Escreva um programa em C que recebe via teclado: **comprimento da circunferência**. O programa deve calcular e imprimir na tela o **diâmetro** e o **raio** da circunferência (veja exemplo abaixo).

Exemplo: Comprimento da circunferência: **36** <**enter**>
Diâmetro: **11.46**
Raio: **5.73**
Continua [S/N]? **N**

comprimento da circunferência = 2 . π . raio $\pi = 3.1416$
diâmetro = 2 . raio

<u>**Observação**</u>: O programa termina quando o usuário digitar 'N' ou 'n' na pergunta: **Continua [S/N]?**

✓ Desenvolva um programa em C que recebe via teclado: **peso** da carne que será vendida (em quilos) e **preço por quilo**. O programa deve calcular e imprimir na tela o **total a pagar**, o **valor pago ao ICMS** (17%) e o **lucro líquido** do açougue.

Exemplo (Tela):

Peso: **3.5** <**enter**>
Preço por Kg (R$): **4.90** <**enter**>
Total a pagar: **17.15**
ICMS (17%): **2.91**
Lucro líquido do açougue (R$): **14.24**
Sair [S/N]? **n**
Peso: **1.5** <**enter**>
Preço por Kg (R$): **9.00** <**enter**>
Total a pagar: **13.5**
ICMS (17%): **2.29**
Lucro líquido do açougue (R$): **11.21**

Sair [S/N]? **S**

- ✓ Escreva um programa em C que recebe via teclado: **tipo de animal** [1] Gado, [2] Eqüinos ou [3] Ovinos, **preço unitário do animal** e **quantidade de animais** comprados. O programa deve calcular e imprimir na tela: **preço total pago** e a **comissão do escritório de remate** (gado → 5%, eqüinos → 7% e ovinos → 3%), conforme exemplo abaixo:

Exemplo (Tela):

Tipo de animal [1] Gado, [2] Eqüinos ou [3] Ovinos: **1** (SEM ENTER)
Preço unitário do animal (R$): **200** <**enter**>
Quantidade de animais: **10** <**enter**>
Preço total pago (R$): **2100.00**
Comissão a pagar (R$): **100.00**
Continua [S/N]? **s**
Tipo de animal [1] Gado, [2] Eqüinos ou [3] Ovinos: **2**
Preço unitário do animal (R$): **1000** <**enter**>
Quantidade de animais: **1** <**enter**>
Preço total pago (R$): **1070.00**
Comissão a pagar (R$): **70.00**
Continua [S/N]? **N**

- ✓ Reescreva o programa anterior recebendo via teclado uma **letra** para o **tipo de animal** [**G**]ado, [**E**]qüinos ou [**O**]vinos, **preço unitário do animal** e **quantidade de animais** comprado. O programa deve calcular e imprimir na tela: **preço total pago** e a **comissão do escritório de remate** (gado → 5%, eqüinos → 7% e ovinos → 3%), conforme exemplo abaixo:

Exemplo (Tela):

Tipo de animal [G]ado, [E]qüinos ou [O]vinos: **g** (SEM ENTER)
Preço unitário do animal (R$): **200** <**enter**>
Quantidade de animais: **10** <**enter**>
Preço total pago (R$): **2100.00**
Comissão a pagar (R$): **100.00**
Continua [S/N]? **s**

- ✓ Escreva um programa em C que recebe via teclado: a **data de hoje** da seguinte forma: **dia**, **mês**, **ano** e a sua **idade**, da seguinte forma: **anos**, **meses** e **dias** vividos. O programa deve calcular e imprimir a data de nascimento no seguinte formato: dd/mm/aaaa.

Exemplo (Tela):

Qual a data de hoje:
Dia: **16** <**enter**>
Mês: **6** <**enter**>
Ano: **2003** <**enter**>
Qual a sua idade:
Anos: **41** <**enter**>
Meses: **4** <**enter**>

Dias: **6** <**enter**>
Data de Nascimento: **10/02/1962**
Continuar [S/N]? **s**

✓ Escreva um programa em C que recebe via teclado um **número inteiro** de 0 à 99. O programa deve imprimir na tela este **número por extenso** (conforme exemplo abaixo). O programa termina quando o usuário digitar 0 (zero).

Exemplo:

Número [0..99]: **23** <**enter**>
Vinte e três
Número [0..99]: **45** <**enter**>
Quarenta e cinco
Número [0..99]: **0** <**enter**>

✓ Escreva um programa em **C** que recebe via teclado: **quantidade de litros** vendidos, **tipo de combustível** ([**A**]lcool, [**G**]asolina ou [**D**]iesel) e o **tipo de pagamento** ([**P**]razo ou [**V**]ista). O programa deve calcular e imprimir na tela: **total à prazo, desconto** e o **total à vista**. O programa termina quando o usuário digitar 'N' ou 'n' na pergunta "Continua [S/N]?".

Tela de execução:

Quantidade de litros? **50** <**enter**>
Tipo de combustível [A]lcool, [G]asolina ou [D]iesel ? **g**
Tipo de pagamento [P]razo ou a [V]ista ? **v**
Total à prazo (R$) : **109.50**
Desconto (R$): **5.48**
Total à vista (R$): **104.02**
Continua [S/N]? **N**

Valores:

Álcool → 1,23
Gasolina → 2,19
Diesel → 1,46

Desconto à vista: 5%

✓ Escreva um programa em C que recebe via teclado duas notas: **nota1 e nota2**. O programa deve imprimir na tela a **média**, o **conceito** do aluno (dado pela tabela abaixo) e a **situação** (aprovado, exame ou reprovado):

Conceito	Média	Situação
A	9,0 à 10,0	Aprovado
B	7,0 à 8,9	Aprovado
C	6,0 à 6,9	Exame
D	0,0 à 5,9	Reprovado

$$\text{Média} = \frac{\text{Nota1} + \text{Nota2}}{2}$$

Exemplo:

Nota1: **7** <**enter**>
Nota2: **8** <**enter**>
Média: **7.5**
Conceito: **B**
Situação: **Aprovado**
Sair [S/N]? **s**

<u>**Observação**</u>: O programa termina quando o usuário digitar ‘S’ ou ‘s’ na pergunta: **Sair [S/N]?**

- ✓ Escreva um programa em C que recebe via teclado uma **temperatura** e o **tipo de conversão** (converter para: [**C**]elsius ou [**F**]ahrenheit). Calcule e imprima na tela a temperatura correspondente a solicitação do usuário, conforme exemplos abaixo:

Exemplo:

Temperatura: **30** <**enter**>
Tipo de conversão (converte para: [C]elsius ou [F]ahrenheit): **F**
Temperatura em **Fahrenheit**: **86**
Continua [S/N]? **S**
Temperatura: **86** <**enter**>
Tipo de conversão (converte para: [C]elsius ou [F]ahrenheit): **C**
Temperatura em **Celsius**: **30**
Continua [S/N]? **n**

Fórmula:

$$9 \cdot \frac{C}{5} = F - 32$$

- ✓ Escreva um programa em C que recebe via teclado: **graus** (0 à 360), **minutos** (0 à 59) e **segundos** (0 à 59). O programa deve calcular e imprimir na tela o **ângulo em graus**, dado pela seguinte fórmula:

$$\text{ângulos em graus} = \text{graus} + \frac{\text{minutos}}{60} + \frac{\text{segundos}}{3600}$$

Exemplo:

Graus: **45** <**enter**>
Minutos: **45** <**enter**>
Segundos: **45** <**enter**>
Ângulo em Graus: **45.76**
Continua [S]im ou [N]ão? **s**
Graus: **45** <**enter**>
Minutos: **10** <**enter**>
Segundos: **15** <**enter**>
Ângulo em Graus: **45.17**
Continua [S]im ou [N]ão? **N**

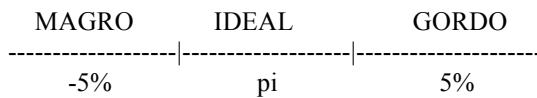
OBSERVAÇÃO: Imprimir mensagens de erro se os valores de entrada estiverem fora da faixa: ERRO: Graus fora da faixa, ERRO: Minutos fora da faixa ou ERRO: Segundos fora da faixa.

- ✓ Escreva um programa em **C** que recebe via teclado: **sexo** ([**M**]asculino ou [**F**]eminino), **altura** e **peso** da pessoa. O programa deve calcular e imprimir na tela: **peso ideal**, **diferença de peso** e **situação** (MAGRO, IDEAL ou GORDO) (conforme exemplo abaixo):

Exemplo:

Sexo [M]asculino ou [F]eminino: **M** (**SEM enter**)
Altura: **1.65** <**enter**>
Peso: **92** <**enter**>
Peso Ideal: **62.0**
Diferença de Peso: **30.0**
Situação: **GORDO**
Sair [S/N]? **s**

PIM = 72,7 x altura – 58
PIF = 62,1 x altura – 44,7

```
   MAGRO          IDEAL               GORDO
-------------------|-------------------|---------------------
       -5%                  pi                   5%
```

<u>**Observação**</u>: O programa termina quando o usuário digitar ‘S’ ou ‘s’ na pergunta: **Sair [S/N]?**

11. Vetores, matrizes e strings

Um vetor é uma coleção de variáveis de mesmo tipo que são referenciadas pelo mesmo nome, utilizando-se um índice para diferencia-los.

Um vetor consiste em locações contíguas de memória, ou seja, os elementos encontram-se em seqüência (contigüidade física). O menor endereço corresponde ao primeiro elemento, e o maior corresponde ao último elemento.

Uma vantagem na utilização de uma vetor é poder armazenar vários valores (elementos), na memória RAM, ao mesmo tempo, permitindo, por exemplo, compará-los e classificá-los.

Exemplo: Vetor unidimensional de inteiros (idades).

Tabela 16: Exemplo de um vetor unidimensional

Índice	Valor
0	24
1	12
2	36
3	41

11.1 - Vetores

Vetor (matriz de uma dimensão - 1D) é um tipo especial de matriz que possui apenas um índice, ou seja, permite armazenar variáveis unidimensionais (permite representar uma tabela).

tipo_dos_dados nome_do_vetor [número_de_elementos];

<u>**Onde**</u>:

tipo_dos_dados: tipo de dado de cada elemento (*char*, *int*, *float*, *double*).
nome_do_vetor: nome da variável que irá representar o vetor
número_de_elementos: número total de elementos do vetor

primeiro elemento: 0
último elemento: número_de_elementos – 1
número de bytes ocupados na memória RAM:
número_de_elementos x quantidade_de_bytes_de_um_elemento

Exemplo: int x[10]; /* 10 elementos: x[0] à x[9] */

primeiro elemento: x[0]
último elemento: x[número_de_elementos – 1], ou seja, x[9]
número de bytes: 10 x 2 = 20 bytes (um inteiro ocupa 2 bytes)

Observação: O programador deve verificar os limites do vetor, pois o compilador C não verifica estes limites, ou seja, o programador pode referenciar qualquer elemento do vetor, inclusive um que não existe. Isto pode causar um grave erro, "*travar*" o programa.

11.2 – Strings

Uma **string** (cadeia de caracteres) é um tipo de dado que consiste de um conjunto de caracteres. Uma *string* pode armazenar qualquer tipo de caracter válido da tabela ASCII.

ASCII: **A**mericam **S**tandard **C**ode for **I**nformation **I**nterchange.
url: http://www.asciitable.com

Em C, uma **string** consiste em um vetor de caracteres finalizado por um zero. Um zero é representado como '\0' ou NULL.

Observação: Na inicialização de uma constante *string* "**teste**" não é necessário acrescentarmos o '\0', pois o compilador faz isso automaticamente.

```
char string[] = "teste";
```

Observação: Note que no exemplo acima, não foi necessário especificar o **número_de_elementos** da **string**. O compilador C interpreta que o tamanho da **string** é a quantidade de caracteres da inicialização (5) mais um (1) para o NULL, totalizandom neste caso, seis (6) elementos. Isto é um problema se quisermos (mais tarde) acrescentar mais caracteres na variável **string**, pois não foi reservado espaço préviamente para estes novos caracteres.

11.3 - Matrizes (Multidimensional)

```
tipo_dos_dados  nome_variável [tamanho_1][tamanho_2]...[tamanho_n];
```

Exemplo: float y[5][5]; /* matriz 2D */

Para acessar o elemento 3, 4 da matriz y, deve-se escrever y[3][4]. Note que o primeiro elemento é y[0][0] e o último elemento é y[4][4]. O total de elementos é 25.

11.4 - Vetor de strings

Para criar um vetor de *strings*, deve-se utilizar uma matriz bidimensional de caracteres, onde o tamanho do índice esquerdo determina o número de *strings* e o tamanho do índice direito especifica o comprimento máximo de cada *string*.

Exemplo: char nome[3][8];

Tabela 17: Exemplo de um vetor de strings

	0	1	2	3	4	5	6	7
0	'U'	'C'	'P'	'e'	'l'	NULL	*lixo*	*lixo*
1	'U'	'C'	'S'	NULL	*lixo*	*lixo*	*lixo*	*lixo*
2	'U'	**'F'**	'P'	'e'	'l'	NULL	*lixo*	*lixo*

Cria um vetor com 3 strings com 7 caracteres + '\0' (NULL) cada uma. Para acessar uma string particular deve-se especificar apenas o índice esquerdo, ou seja, nome[0], nome[1] ou nome[2].

nome[0] ← "UCPel"
nome[1] ← "UCS"
nome[2] ← "UCFel"

Observação: Pode-se acessar também qualquer caracter de qualquer uma das *strings*, isto é feito utilizando os dois índices, como por exemplo, nome[2][1] é caracter 'F'.

11.5 - Inicialização de matrizes e vetores

tipo_dos_dados nome_matriz [tam_1]...[tam_n] = {lista_valores};

lista_valores: lista de constantes separadas por vírgulas que são compatíveis em tipo com o tipo base da matriz.

Exemplo: int i[10] = {0,1,2,3,4,5,6,7,8,9}; /* vetor i – 1D */
ou
int i[] = {0,1,2,3,4,5,6,7,8,9};

Observação: Quando um vetor é declarado e inicializado (ao mesmo tempo) o número de elementos (neste caso 10) pode ser suprimido, ou seja, neste caso é opcional (veja exemplo anterior).

11.6 - Inicialização de um vetor de caracteres

char nome_vetor [tamanho] = "string";

Exemplo: char str[4] = "alo";

0	1	2	3
'a'	'l'	'o'	NULL

11.7 - Inicialização de matrizes multidimensionais

int y[4][2] = { {1,1}, {2,4}, {3,9}, {4,16} };

y[0][0] = 1	y[2][0] = 3
y[0][1] = 1	y[2][1] = 9
y[1][0] = 2	y[3][0] = 4
y[1][1] = 4	y[3][1] = 16

11.8 - Inicialização de vetores e matrizes sem tamanho

Na inicialização de uma matriz (ou vetor), se não for especificado seu tamanho, então o compilador C cria uma matriz (ou vetor) grande o suficiente para conter todos os inicializadores presentes.

Exemplo: char s[] = "UCPel"; /* s ocupa 6 bytes */

Programa exemplo (23): O programa permite armazenar ***n*** nomes e idades em dois vetores.

```
#include <stdio.h>
#include <conio.h>

#define MAX 10

void main(void)
{
 char nome[MAX][41];
 int idade[MAX];
 int i, n;
 char ch;

 i = 0;
 do {
     clrscr();
     printf("Nome: ");
     scanf("%s", nome[i]);                    /* entrada de um nome */
     flushall();                              /* limpa o buffer de entrada */
     printf("Idade: ");
     scanf("%d", &idade[i]);                  /* entrada de uma idade */
     i++;
     printf("Continua [S/N]? ");
     do {
          ch = getch();
        } while (ch != 'S' && ch != 's' && ch != 'N' && ch != 'n');
    } while (ch != 'N' && ch != 'n' && i < MAX);
 n = i - 1;                                   /* número de elementos */
 clrscr();
 for (i = 0; i <= n; i++)
     printf("| %s | %d | \n", nome[i], idade[i]);
 getch();
}
```

Programa exemplo (24): O programa realiza a soma de duas matrizes (A e B) bidimensionais, gerando uma matriz resultante C.

```
#include <stdio.h>
#include <conio.h>

#define MAX 10

void main(void)
{
 float a[MAX][MAX], b[MAX][MAX], c[MAX][MAX];
 int col,lin,j,m,n;

 clrscr();
 printf("Informe a ORDEM da MATRIZ: (mxn)\n");
 do {
      printf("Número de linhas (m): ");
      scanf("%d",&m);
     } while (m < 1 || m > MAX);              /* m de 1 à 10 */
 do {
      printf("Número de colunas (n): ");
      scanf("%d",&n);
     } while (n < 1 || n > MAX);              /* n de 1 à 10 */
 for (lin = 1;lin <= m;lin++)
     for (col = 1;col <= n;col++)
         {
          printf("A[%d,%d] = ",lin,col);
          scanf("%d",&a[lin][col]);
          printf("B[%d,%d] = ",lin,col);
          scanf("%d",&b[lin][col]);
         }
 printf("\n");
 for (lin = 1;lin <= m;lin++)
     for (col = 1;col <= n;col++)
         {
          c[lin][col] = a[lin][col] + b[lin][col];
          printf("C[%d,%d] = %d\n",lin,col,c[lin][col]);
         }
 getch();
}
```

11.9 - Classificação de dados ou ordenação (sort)

Para exemplificar melhor as variáveis do tipo vetor, abaixo são mostrados dois tipos de ordenação, também chamado **sort** (classificação de dados):

Programa exemplo (25): O programa classifica os nomes digitados pelo usuário.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
```

```
#define QUANT 50

void main(void)
{
 char nome[QUANT][40];
 char temp[40];
 int i, j, n;
 char tecla;

 n = -1;
 do {
      n++;
      clrscr();
      printf("Nome: ");
      gets(nome[n]);                    /* gets – permite a entrada de um nome */
      printf("Continua [S/N]? ");
      do {
           tecla = toupper(getche());          /* toupper – converte o caracter para maiúsculo */
          } while (!strchr("SN", tecla));  /* strchr – verifica se um caracter pertence a string */
    } while (tecla != 'N' && n < QUANT);

  for (i = 0; i <= n-1; i++)
     for (j = i+1; j <= n; j++)
            if ((strcmp(nome[i], nome[j])) > 0)       /* strcmp – permite comparar strings */
              {
               strcpy(temp, nome[i]);                 /* strcpy – permite copiar strings */
               strcpy(nome[i], nome[j]);
               strcpy(nome[j], temp);
              }
 printf("\nNomes ORDENADOS:");
 for (i = 0; i <= n; i++)
     printf("Nome: %s\n", nome[i]);
  getch();
}
```

Observação: As funções **strcpy**, **strcmp** e **strchr** são discutidas na próxima seção.

Programa exemplo (26): O programa utiliza um método de sort chamado ***bubble sort*** (método da bolha) para classificar nomes.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>

#define TRUE   !0
#define FALSE  0

#define QUANT 50

void main(void)
{
 char nome[QUANT][40];
```

```
 char temp[40], tecla;
 int i, k, n;
 int sort;

 n = 0;
 do {
      clrscr();
      printf("Nome: ");
      gets(nome[n]);                          /* entrada de um nome */
      n++;
      printf(" Continua [S/N]? ");
      do {
           tecla = getche();
         } while (!strchr("SsNn", tecla));
     } while (strchr("Nn", tecla) && n < QUANT);
 n = n - 1;                                   /* n → número de elementos */

k = n;
 do {
       sort = FALSE;
       for (i = 0; i < k; i++)
            if ((strcmp(nome[i],nome[i+1])) > 0)
              {
               strcpy(temp, nome[i]);         Buble sort
               strcpy(nome[i], nome[i+1]);
               strcpy(nome[i+1], temp);
               sort = TRUE;
              }
      k--;
    } while (sort);

 printf("\nNomes ORDENADOS:\n");
 for (i = 0; i <= n; i++)
    printf("Nome: %s\n", nome[i]);
 getch();
}
```

11.10 - Lista de exercícios (vetores)

✓ Escreva um programa em C que recebe via teclado um conjunto de letras (máximo 20). Armazene todas as letras em um vetor (**letras**) até que o usuário digite um **ESC** (código 27). Logo após copie todas as letras (***em ordem inversa***) para outro vetor (**inverso**). Ao final imprima os dois vetores.

letras

0	'L'
1	'I'
2	'M'
3	'A'

inverso

0	'A'
1	'M'
2	'I'
3	'L'

Exemplo:

Letra: **L**

Letra: **I**

Letra: **M**

Letra: **A**

Letra: <**esc**>

LIMA

AMIL

- ✓ Escreva um programa em C que recebe via teclado: **número de idades** (máximo 50) e as respectivas **idades**. Armazene todas as idades em um vetor (**idade**). Logo após a entrada de todas as idades, o programa deve receber via teclado: **idade para consulta**. O programa deve imprimir na tela, o número de idades **antes** da idade de consulta e o número de idades **depois** da idade de consulta.

Exemplo:

Número de idades: **6** <**enter**>
Idade: **30** <**enter**>
Idade: **60** <**enter**>
Idade: **10** <**enter**>
Idade: **50** <**enter**>
Idade: **20** <**enter**>
Idade: **40** <**enter**>
Idade para consulta: **50** <**enter**>
Antes: **3**
Depois: **2**
Continua [S/N]? **n**

- ✓ Escreva um programa em C que recebe via teclado um conjunto de **números inteiros** (máximo 50). Armazene todos os números inteiros em um vetor até que o usuário digite 0 (zero). Logo após permita ao usuário consultar um número informando o seu **valor**. O programa deve imprimir na tela a **posição do número no vetor** ou **ERRO: Número não encontrado** (veja exemplos abaixo):

Exemplo:

Número: **50** <**enter**>
Número: **30** <**enter**>
Número: **20** <**enter**>
Número: **10** <**enter**>
Número: **40** <**enter**>
Número: **0** <**enter**>
Valor: **20** <**enter**>
Posição no vetor: **2**
Valor: **40** <**enter**>
Posição no vetor: **4**
Valor: **60** <**enter**>
ERRO: Número não encontrado
Valor: 0 <**enter**>

Observação: O programa termina quando o usuário digitar 0 (zero).

- ✓ Escreva um programa em C que recebe via teclado "n" **conceitos** (**A**, **B**, **C**, **D** e **E**) (máximo 25) até que o usuário digite **ESC**. Armazene todos os conceitos em um vetor (**conceito**). Imprima na tela o número de alunos: **aprovados** (A, B e C), **reprovados** (D) e os **infreqüentes** (E).

Exemplo:

Conceito: **B**
Conceito: **A**
Conceito: **E**
Conceito: **B**

conceito

0	'B'
1	'A'
2	'E'
3	'B'
4	'D'
5	'C'
6	'A'

Conceito: **D**
Conceito: **C**
Conceito: **A**
Conceito: **E**
Conceito: <**esc**>
4 Aprovado(s)
1 Reprovado(s)
3 Infreqüente (s)

✓ Escreva um programa em C que recebe via teclado "n" (máximo 50) **nomes** (máximo 80 letras). A entrada dos nomes termina quando o usuário digitar apenas <**enter**>. Logo após a entrada de todos os nomes o programa deve permitir a entrada via teclado de uma **letra**. O programa deve imprimir na tela todos os nomes que começam com a letra especificada pelo usuário. O programa termina quanto o usuário digitar <**esc**> na entrada da letra (conforme exemplos abaixo):

Exemplo: Nome: **Paulo** <**enter**>
Nome: **Roberto** <**enter**>
Nome: **Renato** <**enter**>
Nome: **Pedro** <**enter**>
Nome: **Fabio** <**enter**>
Nome: <**enter**>
Letra: **R**
Nome: **Roberto**
Nome: **Renato**
Letra: **P**
Nome: **Paulo**
Nome: **Pedro**
Letra: **T**
Letra: <**esc**>

✓ Escreva um programa em C que recebe via teclado "n" (máximo 30) **nomes** (máximo 40 letras) e **idades**. A entrada dos dados termina quando o usuário digitar 'N' ou 'n' na pergunta "Continua [S/N]?". Logo após a entrada de todos os dados o programa deve imprimir na tela todos os nomes e idades desde o mais velho até o mais novo.

Exemplo: Nome: **Ana** <**enter**>
Idade: **12** <**enter**>
Continua [S/N]? **s**
Nome: **Beatriz** <**enter**>
Idade: **13** <**enter**>
Continua [S/N]? **s**
Nome: **Carla** <**enter**>
Idade: **14** <**enter**>
Continua [S/N]? **N**
Carla **14**
Beatriz **13**
Ana **12**

12. Manipulação de strings

As funções **strcpy**, **strcmp**, **strcat** são necessárias pois uma string nada mais é do que um vetor de caracteres, ou seja, não se pode atribuir vários valores (ao mesmo tempo) para os elementos de um vetor. Isto só pode ser feito quando o vetor é declarado e inicializado ou um de cada vez.

Exemplo: char s[] = “UCPel”; /* **correto** */

ou

```
s[0] = ‘U’;          /* correto – elemento por elemento */
s[1] = ‘C’;
s[2] = ‘P’;
s[3] = ‘e’;
s[4] = ‘l’;
s[5] = NULL;
```

ou

```
s = “UCPel”;         /* incorreto – erro GRAVE em C */
```

12.1 - strcpy

A função **strcpy** (cópia de string) pré-definida do C que permite copiar uma string para outra ou inicializar uma string.

Sintaxe: char ***strcpy** (char *destino, const char *origem);
Prototype: string.h

Funcionamento: A *string* de origem é copiada para a *string* destino.

Programa exemplo (27): Os programas abaixo mostram como copiar caracteres para uma *string*.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
 char erro[] = "Arquivo não Existe\n";

 printf("ERRO: %s",erro);
 getch();
}
```

ou

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 char erro[20];
```

```
 strcpy(erro,"Arquivo não existe\n");
 printf("ERRO: %s",erro);
 getch();
}
```

12.2 - strcmp

A função **strcmp** (comparação de duas strings) pré-definida do C que permite comparar duas strings.

s1 é maior que s2 → resultado > 0
s1 é menor que s2 → resultado < 0
s1 é igual a s2 → resultado == 0
s1 é diferente de s2 → resultado != 0

Sintaxe: int **strcmp** (const char *s1, const char *s2);
Prototype: string.h

Programa exemplo (28): O programa compara duas *strings*.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
 char s1[41],s2[41];

 clrscr();
 printf("String 1: ");
 gets(s1);
 printf("String 2: ");
 gets(s2);
 if (strcmp(s1,s2) == 0)
    printf("String 1 é IGUAL a String 2");
 else
    if (strcmp(s1,s2) > 0)
       printf("String 1 é MAIOR a String 2");
    else
       printf("String 1 é MENOR a String 2");
 getch();
}
```

12.3 - strcat

A função **strcat** (concatenação de duas strings) pré-definida do C que permite a concatenação de uma string no final da outra string.

Sintaxe: char ***strcat** (char *destino, const char *origem);
Prototype: string.h

Funcionamento: A *string* origem é copiado para o final da *string* destino.

Programa exemplo (29): O programa cópia e concatena caracteres a uma *string* resultando na palavra "Pelotas".

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
 char s1[] = "Pel ",s2[] = "ota ",s3[] = "s";
 char sr[15] = "";

 clrscr();
 printf("%s\n",s1);
 printf("%s\n",s2);
 printf("%s\n",s3);
 strcpy(sr,s1);           /* sr = "Pel" */
 strcat(sr,s2);           /* sr = "Pelot" */
 strcat(sr,s3);           /* sr = "Pelotas" */
 printf("%s\n",sr);
 getch();
}
```

12.4 - strlen

A função **strlen** (comprimento de uma string) pré-definida do C que retorna o comprimento de uma *string*, ou seja, a quantidade de caracteres que a *string* possui no momento.

Observação: O NULL não é contado.

Sintaxe: int **strlen** (const char *s);
Prototype: string.h

Programa exemplo (30): O programa imprime na tela a quantidade de caracteres de uma variável string, neste caso, a variável nome.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
 char nome[41];

 clrscr();
 printf("Qual o seu nome: ");
 gets(nome);
 printf("%s seu nome tem %d caracteres\n", nome, strlen(nome));
```

```
 getch();
}
```

12.5 – strchr

A função **strchr** (Verifica se um caracter pertence a uma string) pré-definida do C que verifica se um caracter (**chr**) pertence a uma string (**str**).

Sintaxe: int **strchr** (const char *s, char ch);
Prototype: string.h

Programa exemplo (31): O programa imprime na tela o número de caracteres de um nome.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
 char nome[41];
 char ch;

 do {
      clrscr();
      printf("Qual o seu nome: ");
      gets(nome);
      printf("Seu nome tem %d caracteres\n", strlen(nome));
      printf("Continua [S]im ou [N]ão ?");
      do {
           ch = getche();
         } while (!strchr("SsNn",ch));
    } while (strchr("Ss",ch));
}
```

12.6 – Lista de exercícios (strings)

✓ Escreva um programa em C que recebe via teclado um **nome** (máximo 256 caracteres). Logo após a entrada do nome imprima: **número de letras maiúsculas**, **número de letras minúsculas**, **número de vogais** e o **número de consoantes**, conforme exemplo abaixo:

Exemplo: Nome: **Universidade Católica de Pelotas** <**enter**>
(**3**) Letras maiúsculas
(**26**) Letras minúsculas
(**14**) Vogais
(**15**) Consoantes

✓ Escreva um programa em C que recebe via teclado uma **palavra** (máximo 40 caracteres) e uma **letra**. Logo após a entrada do nome e da letra imprima **o**

número de letras que existe no nome ou **ERRO: Não existe a letra (?) na palavra (?)**, conforme exemplo abaixo:

Exemplo:

Palavra: **Luzzardi** <**enter**>
Letra: **z**
2 letra(s)
Continua [S]im ou [N]ão? **s**
Palavra: **Luzzardi** <**enter**>
Letra: **w**
ERRO: Não existe a letra (w) na palavra (Luzzardi)
Continua [S]im ou [N]ão? **N**

<u>**Observação**</u>: O programa deve ser encerrado quando o usuário digitar “N” ou “n” na pergunta: **Continua [S]im ou [N]ão?**.

✓ Escreva um programa em C que recebe via teclado uma **palavra** (máximo 20 letras) e uma **posição**. O programa deve imprimir na tela, a letra **antecessora**, a **letra** (da referida posição) e a letra **sucessora**, conforme exemplo abaixo:

Exemplo:

Palavra: **Universidade** <**enter**>
Posição: **7** <**enter**>
Antecessora: **s**
Letra: **i**
Sucessora: **d**

OBSERVAÇÃO: O programa deve imprimir na tela as seguintes mensagens de erro, se for o caso: **Letra antecessora não existe**, **Letra sucessora não existe** ou **Posição inválida**.

✓ Escreva um programa em C que recebe via teclado um **nome** (máximo 80 letras). O programa deve imprimir, na tela, as **palavras** do nome <u>**em ordem inversa, uma por linha**</u>, conforme exemplo abaixo:

Exemplo:

Nome: **Paulo Roberto Gomes Luzzardi** <**enter**>
Luzzardi
Gomes
Roberto
Paulo

✓ Escreva um programa em C que recebe via teclado um **nome** (máximo 60 caracteres). Logo após a entrada do nome o programa deve imprimir (EM LETRA MAIÚSCULA) o **sobrenome da pessoa**, conforme exemplos abaixo:

Exemplo:

Nome: **Paulo Roberto Gomes Luzzardi** <**enter**>
Sobrenome: **LUZZARDI**
Sair [S/N]? **N**
Nome: **Renato Souza** <**enter**>
Sobrenome: **SOUZA**
Sair [S/N]? **s**

<u>**Observação**</u>: O programa termina quando o usuário digitar ‘S’ ou ‘s’ na pergunta: **Sair [S/N]?**

✓ Escreva um programa em C que recebe via teclado um **nome** (máximo 80 caracteres). Logo após a entrada do nome o programa deve imprimir na tela:

sobrenome, **primeiro nome** e **demais nomes abreviados**, conforme exemplos abaixo:

Exemplo: Nome: **Paulo Roberto Gomes Luzzardi** <**enter**>
Autor: **Luzzardi, Paulo R. G.**
Sair [S/N]? **N**
Nome: **Renato Lima Souza** <**enter**>
Autor: **Souza, Renato L.**
Sair [S/N]? **s**

Observação: O programa termina quando o usuário digitar 'S' ou 's' na pergunta: **Sair [S/N]?**

✓ Escreva um programa em C que recebe via teclado o nome de um **estado** (máximo 80 caracteres). Logo após a entrada do nome do estado imprima: a **sigla** do estado (2 letras maiúsculas), conforme exemplos abaixo:

Exemplo: Estado: **Rio Grande do Sul** <**enter**>
Sigla: **RS**
Estado: **são paulo** <**enter**>
Sigla: **SP**
Estado: **rio de janeiro** <**enter**>
Sigla: **RJ**
Estado: <**enter**>

Observação: O programa encerra quando o usuário digitar apenas <**enter**> na entrada do nome do estado.

✓ Escreva um programa em C que recebe via teclado uma **password** (senha – máximo 8 dígitos). Na entrada da senha deve ser exibido na tela um asterisco (*) para cada letra digitada. Quando o usuário teclar <**enter**> (ou digitar 8 dígitos) o programa deve imprimir na tela a senha digitada.

Exemplo: Password: *********** <**enter**>
Senha digitada: **pelotas**
Sair [S/N]? **s**

Observação: O programa deve ser encerrado quando o usuário digitar 'S' ou 's' na pergunta: **Sair [S/N]?**.

✓ Escreva um programa em C que recebe via teclado uma **palavra** (máximo 20 caracteres), **início** e **fim**. Logo após a entrada de todos os dados imprima a **string resultante** ou **ERRO: Fim inválido** ou **Início inválido**, conforme exemplos abaixo:

Exemplo: Palavra: **universidade** <**enter**>
Início: **7** <**enter**>
Fim: **11** <**enter**>
String resultante: **idade**
Continua [S/N]? **s**
Palavra: **eletricidade** <**enter**>
Início: **7** <**enter**>
Fim: **15** <**enter**>

```
ERRO: Fim Inválido
Continua [S/N]? N
```

Observação: O programa termina quando o usuário digitar 'N' ou 'n' na pergunta: **Continua [S/N]?**.

13. Funções definidas pelo programador

C permite que o programador crie e utilize suas próprias funções.

Forma Geral:

```
tipo_do_retorno nome_da_função (parâmetros)
tipo_dado_base parâmetros;
{
 tipo_dado_base variáveis;                    /* definição das variáveis locais */

 corpo da função;

 return();
}
```

ou

```
tipo_do_retorno nome (tipo_dado_base parâmetros)
{
 tipo_dado_base variáveis;

 corpo da função;

 return();
}
```

tipo_do_retorno: Especifica o tipo de dado do retorno da função. O retorno da função é feita pelo comando ***return*** (valor).

parâmetros: É uma lista, separada por vírgulas, com os nomes das variáveis (e seus tipos) que receberão os argumentos quando a função for chamada.

Observação: O tipo default de uma função é **int**, ou seja, se não for especificado o tipo da função o compilador assume **int**.

Função procedural: É um tipo especial de função que não possui retorno, ou seja, é simplesmente um procedimento. Uma função deste tipo é **void**.

Exemplo:

```
void imprime_string (int coluna, int linha, char *mensagem)
{
 gotoxy(coluna, linha);
```

```
 printf("%s", mensagem);
}
```

Chamada da função: imprime_string(10, 7, "Calculadora");

Programa exemplo (32): O programa possui uma função que calcula o inverso 1/x.

```
#include <stdio.h>
#include <conio.h>

float inverso (float x);

void main(void)
{
 float inv, x;

 clrscr();
 printf("x: ");
 scanf("%f",&x);
 inv = inverso (x);                    /* chamada da função inverso */
 printf("Inverso = %.2f\n", inv);
 getch();
}

/* ---------------------------- Função definida pelo programador */

float inverso (float x)
{
 float i;

 i = (float) 1 / x;
 return(i);
}
```

ou

```
float inverso (x)
float x;
{
 return( (float) 1 / x );
}
```

13.1 - Valores de retorno

Todas as funções, exceto aquelas que são declaradas como sendo do tipo **void**, devolvem um valor. O valor é devolvido (retornado) pela função através do comando **return**.

Normalmente são escritas três tipos de funções:

a) Funções que efetuam operações com os parâmetros e retornam um valor com base nas operações.

Programa exemplo (33): O programa calcula e imprime na tela o valor da potência x^y.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

float potencia (float x, int y);

void main(void)
{
 float base,resp;
 int expoente;

 clrscr();
 printf("Base: ");
 scanf("%f",&base);
 printf("Expoente: ");
 scanf("%d",&expoente);
 resp = potencia(base,expoente);                    /* chamada da função potencia */
 printf("Potencia = %7.2f\n",resp);
 getch();
}

/* ---------------------------- Função definida pelo programador */

float potencia (float x, int y)
{
 float valor;

 valor = exp ( log (x ) * y );
 return(valor);
}
```

b) <u>Funções que manipulam informações e retornam um valor que simplesmente indica o sucesso ou o fracasso da manipulação</u>.

Programa exemplo (34): O programa calcula e verifica o determinante de uma equação de segundo grau.

```
#include <stdio.h>
#include <conio.h>

int verifica_determinante (float a, float b, float c);

void main(void)
{
 float a, b, c;
 int retorno;

 clrscr();
 printf("a = ");
 scanf("%f",&a);
 printf("b = ");
 scanf("%f",&b);
```

```
 printf("c = ");
 scanf("%f",&c);
 retorno = verifica_determinante(a,b,c);
 if (retorno == 0)
    printf("Determinante ZERO\n");
 else
    if (retorno > 0)
       printf("Determinante POSITIVO\n");
    else
       printf("Determinante NEGATIVO\n");
 getch();
}

/* ------------------------------------------- função definida pelo programador */

int verifica_determinante (float a, float b, float c)
{
 float det;

 det = b * b - 4 * a * c;
 if (det == 0)
   return(0);
 else
    if (det > 0)
       return(1);
    else
       return(-1);
}
```

c) <u>Funções que não retornam nenhum valor, ou seja, são puramente procedimentos</u>.

Programa exemplo (35): O programa possui uma função que limpa toda a tela.

```
#include <stdio.h>
#include <conio.h>

void limpa_tela (void);

void main(void)
{
 limpa_tela();
 getch();
}

/* -------------------------------------------- função definida pelo programador */

void limpa_tela (void)
{
 int c, l;

 for (l = 1; l <= 25; l++)
    for (c = 1; c <= 80; c++)
       {
        gotoxy (c, l);
```

```
      printf("%c",32);                    /* 32 código do caracter espaço */
     }
}
```

13.2 - Passagem de parâmetros por valor

Forma de chamada de uma função onde o valor do argumento é apenas copiado para o parâmetro formal da função. Portanto, alterações feitas nos parâmetros não terão efeito nas variáveis utilizadas para chamá-la.

Programa exemplo (36): O programa possui uma função que pinta uma parte da tela.

```
#include <stdio.h>
#include <conio.h>

void pinta_janela (int ci, int li, int cf, int lf, int cor);

void main(void)
{
 textbackground(WHITE);
 clrscr();
 pinta_janela(20, 5, 40, 20, BLUE);
 getch();
}

void pinta_janela (int ci, int li, int cf, int lf, int cor)
{
 window (ci, li, cf, lf);
 textbackground(cor);
 clrscr();
 window(1, 1, 80, 25);
}
```

Atenção: Os parâmetros da função recebem, respectivamente: **ci ← 20**, **li ← 5**, **cf ←40** e **lf ←20**.

13.3 - Passagem de parâmetros por referência

Forma de chamada de uma função onde o endereço do argumento é passado como parâmetro. Significa que as alterações feitas nos parâmetros afetarão a variável utilizada para chamar a função.

Programa exemplo (37): O programa tem uma função que troca o valor de duas variáveis.

```
#include <stdio.h>
#include <conio.h>

void troca (int *x, int *y);

void main(void)
{
 int a, b;
```

```
 clrscr();
 printf("a = ");                          /* imagine que o usuário digitou 3 */
 scanf("%d",&a);
 printf("b = ");
 scanf("%d",&b);                          /* imagine que o usuário digitou 4 */
 printf("a = %d | b = %d\n", a, b);       /* foi impresso na tela a = 3 | b = 4 */
 troca (&a,&b);
 printf("a = %d | b = %d\n", a, b);       /* foi impresso na tela a = 4 | b = 3 */
 getch();
}

void troca (int *x, int *y)               /* x e y são ponteiros */
{
int temp;

temp = *x;
*x = *y;
*y = temp;
}
```

Atenção: Os parâmetros da função recebem, respectivamente: **x ← &a** e **y ← &b**.

13.4 - Funções que devolvem valores não-inteiros

Todas as funções que devolvem valores não-inteiros devem ter seu tipo de retorno declarado.

tipo_do_retorno nome_da_função (tipo_dado_base parâmetros);

Programa exemplo (38): O programa calcula e imprime na tela a divisão de dois valores.

```
#include <stdio.h>
#include <conio.h>

float divisao (int x, int y);

void main(void)
{
 int x, y;
 float resposta;

 clrscr();
 printf("x = ");
 scanf("%d",&x);
 printf("y = ");
 scanf("%d",&y);
 resposta = divisao ( x, y);
 printf("Divisão = %7.2f\n",resposta);
 getch();
}

/* ---------------------------- Função definida pelo programador */
```

```
float divisao (int x, int y)
{
 return( (float) x / y );
}
```

13.5 - Argumentos argc e argv do main

A função **main** possui dois argumentos **argc** e **argv** intrínsecos utilizados para receber parâmetros da linha de comando do DOS (Sistema Operacional).

argc - contém o número de argumentos na linha de comando.
argv - ponteiro para uma matriz (2D) de caracteres (vetor de strings).

Programa exemplo (39): O programa recebe parâmetros do Sistema Operacional (uma palavra qualquer) e imprime a palavra em sentido inverso. O programa é compilado e executado no Sistema Operacional da seguinte forma:

Execução pelo DOS: A:\>inverte pelotas <**enter**>

Resultado na tela: *satolep*

Programa recebe:

```
argc ← 2
argv[0] ← “A:\inverte.exe”
agv[1] ← “pelotas”
```

```
/* inverte.c */            /* obrigatoriamente este programa deve ser gravado como inverte.c */
                           /* depois de compilador será gerado o executável inverte.exe */
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main ( int argc, char *argv[] )
{
 int i, n;

 if (argc != 2)
    printf("Sintaxe: INVERTE <palavra>");
 else
    {
     n = strlen(argv[1]);
     for (i = n-1; i >= 0; i--)
           printf("%c", argv[1][i]);
    }
}
```

Programa exemplo (40): O programa recebe parâmetros pelo Sistema Operacional (conjunto de caracteres) e ordena (coloca em ordem alfabética), imprimindo-os a seguir.

Execução pelo DOS: A:\>ordena dbeacgf <**enter**>

Resultado na tela: abcdefg

Programa recebe:

```
argc ← 2
argv[0] ← "A:\ordena.exe"
agv[1] ← "dbeacgf"
```

```
/* ordena.c */

#include <stdio.h>
#include <conio.h>
#include <string.h>

void main ( int argc, char *argv[] )
{
 int i, j;
 int n_car;
 char temp;

 if (argc != 2)
    printf("Sintaxe: ORDENA <palavra>");
 else
    {
     n_car = strlen(argv[1]);
     for (i = 0; i < n_car-1; i++)
         for (j = i+1; j < n_car; j++)
             if (argv[1][i] > argv[1][j])
               {
                temp = argv[1][i];
                argv[1][i] = argv[1][j];
                argv[1][j] = temp;
              }
      for (i = 0 ; i < n_car ; i++)
              printf("%c",argv[1][i]);
    }
}
```

13.6 - Recursividade

Uma função é recursiva, se esta, fizer uma chamada a si própria.

Programa exemplo (41): Este programa calcula o **fatorial** de um número recursivamente.

```
#include <stdio.h>
#include <conio.h>

long int fatorial (unsigned int n);

void main (void)
{
```

```
 unsigned int n;
 long unsigned int resposta;

 clrscr();
 do {
       printf("Valor = ");
       scanf("%d",&n);
      } while (n < 0 || n > 15);
  resposta = fatorial (n);
  printf("Fatorial de [%d] = [%d]\n", n, resposta);
  getch();
 }

long int fatorial (unsigned int n)
{
 long int resp, valor;

 if (n <= 0)
    return (0);
 if (n == 1)
    return (1);
 valor = fatorial (n - 1);
 resp = valor * n;
 return (resp);
}
```

13.7 - Lista de exercícios (funções)

✓ Escreva em C a função **PALAVRAS**. A função recebe uma string (**nome**) e retorna o **número de palavras do nome** (veja exemplo abaixo):

```
#include <stdio.h>
#include <conio.h>

______________ PALAVRAS( ____________________ );

void main(void)
{
 char nome[256];
 int n;

 clrscr();
 printf("Nome: ");
 gets(nome);
 n = PALAVRAS(nome);
 printf("Seu nome tem %d palavra(s)\n", n);
 getch();
}
```

Exemplo:

Nome: **Paulo Roberto Gomes Luzzardi** <**enter**>
Seu nome tem **4** palavra(s)

Observação: Não utilizar a função **strlen** do Turbo C.

✓ Escreva em C a função **VERIFICA_DATA**. A função recebe uma string (**data**, no formato: dd/mm/aaaa) e devolve via parâmetros: **dia**, **mês** e **ano**. A função

retorna: (**1**) se o dia for inválido, (**2**) se o mês for inválido, (**3**) se o ano for inválido, (**4**) formato inválido e (**5**) se a data estiver correta.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>

______________ VERIFICA_DATA ( ____________________ );

void main (void)
{
 char data[11];
 int dia, mes, ano, erro;

 clrscr();
 printf("Data [dd/mm/aaaa]: ");
 scanf("%s", data);
 erro = VERIFICA_DATA(data, &dia, &mes, &ano);
 if (erro == 1)
        printf("ERRO: Dia inválido\n");
 else
        if (erro == 2)
                printf("ERRO: Mês inválido\n");
        else
                if (erro == 3)
                        printf("ERRO: Formato inválido\n");
                else
                        if (erro == 4)
                              printf("ERRO: Ano Inválido\n");
                        else
                             {
                              printf("Dia: %d\n", dia);
                              printf("Mês: %d\n", mes);
                              printf("Ano: %d\n", ano);
                             }
 getch();
}
```

Exemplo:

Data [dd/mm/aaaa]: **11/07/2002** <**enter**>
Dia: **11**
Mês: **07**
Ano: **2002**

Valores Válidos:

Dia [1..31]
Mês [1..12]
Ano [2000.. 2999]

✓ Escreva em C a função **PREENCHE**. A função recebe: **coluna** (c), **linha** (l), **número de caracteres** (n) e **caracter** (ch). A função deve imprimir na tela, na posição (**c**, **l**), **n** caracteres **ch**.

```
#include <stdio.h>
#include <conio.h>
_____________________ PREENCHE ( ____________________ );

void main (void)
{
 clrscr();
 PREENCHE(3,10,5,'#');
 getch();
}
```

```
           |                                  |
linha 10→  |  #####                           |
           |                                  |
```

✓ Escreva em C a função **GERA_STRING**. A função recebe via parâmetros **número de caracteres** (n) e **caracter** (ch). A função devolve na variável **s**, **n** caracteres **ch**.

```
#include <stdio.h>
#include <conio.h>
____________________ GERA_STRING ( ________________ );

void main (void)
{
 char s[10];                                              s
                                          +-----------------------------------+
 clrscr();                                | 0 | 1 |2 | 3 | 4 |   5    |
 GERA_STRING(5, '#', s);                  |-------------------------------------|
 gotoxy(3, 10);                           | '#' | '#' | '#' | '#' | '#' | NULL |
 printf("%s", s);                         +-----------------------------------+
 getch();
}
```

✓ Escreva em C a função **VERIFICA_QUADRANTE**. A função recebe um valor para **x** e um valor para **y** e retorna o **número do quadrante** (1,2,3 ou 4).

```
#include <stdio.h>
#include <conio.h>
____________________ VERIFICA_QUADRANTE ( ________________ );

void main (void)                                         Quadrantes
{
 int x, y, n;                                                |
                                                             |
 clrscr();                                          2º       |       1º
 printf("x = ");                                             |
 scanf("%d", &x);                             --------------------------------------
 printf("y = ");                                             |
 scanf("%d", &y);                                   3º       |       4º
 n = VERIFICA_QUADRANTE(x, y);                               |
 printf("Quadrante: %d\n", n);                               |
getch();
}
```

✓ Escreva a função: **final_da_placa**. A função recebe uma placa de automóvel (placa) no formato: xxx9999 e retorna o último dígito da placa.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main (void)
{
 char placa[8];
 int final;

 clrscr();
 printf("Qual a placa de seu carro [xxx9999]: ");
```

```
 scanf("%s", placa);
 final = final_da_placa(placa);
 printf("Final da Placa é: %d\n", final);
 getche();
}
```

- ✓ Escreva a função: **VOGAIS.** A função recebe uma **string** (nome) e retorna a quantidade de **vogais** da string.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main (void)
{
 char nome[256];
 int vogais;

 clrscr();
 printf("Nome: ");
 gets(nome);
 vogais = VOGAIS(nome);
 printf("Vogais: %d\n", vogais);
 getche();
}
```

- ✓ Escreva a função: **HIPOTENUSA**. A função recebe o cateto adjacente (b) e o cateto oposto (a) e retorna o valor da hipotenusa dado pela seguinte fórmula:

Fórmula: $h^2 = a^2 + b^2$

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main (void)
{
 float a, b, h;

 clrscr();
 printf("Cateto Adjacente: ");
 scanf("%f", &b);
 printf("Cateto Oposto: ");
 scanf("%f", &a);
 h = HIPOTENUSA(a, b);
 printf("Hipotenusa: %.2f\n", h);
 getche();
}
```

- ✓ Escreva em C a função **CONTA**. A função recebe uma string (**nome**) e devolve via parâmetros: **número letras maiúsculas** e o **número letras minúsculas**. A função retorna o **total de letras do nome** (veja exemplo abaixo).

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

_______________ CONTA ( _____________________ );

void main (void)
{
 char nome[256];
 int n, maiusculas, minusculas;

 clrscr();
 printf("Nome: ");
 gets(nome);
 n = CONTA(nome, &maiusculas, &minusculas);
 printf("Maiúsculas: %d\n", maiusculas);
 printf("Minúsculas: %d\n", minusculas);
 printf("Total de letras: %d\n", n);
 getch();
}
```

Exemplo:

Nome: **Paulo Roberto Gomes Luzzardi** <**enter**>
Maiúsculas: **4**
Minúsculas: **21**
Total de letras: **25**

- ✓ Escreva em C a função **REAJUSTE**. A função recebe o valor do **salário** e o **índice** de reajuste e retorna o **salário atualizado**.

```
#include <stdio.h>
#include <conio.h>
____________________ REAJUSTE ( ________________ );

void main (void)
{
 float salario, indice, sal;

 clrscr();
 printf("Salário (R$):  ");
 scanf("%f", &salario);
 printf("Índice de Reajuste: ");
 scanf("%f", &indice);
 sal = REAJUSTE(salario, indice);
 printf("Salário Atualizado (R$): %.2f\n", sal);
 getch();
}
```

Exemplo:

Salário (R$): **1000** <**enter**>
Índice de Reajuste: **10** <**enter**>
Salário Atualizado (R$): **1100**

- ✓ Escreva em C a função **CENTRALIZA**. A função recebe: **linha**, **mensagem** e **cor**. A função deve imprimir na tela a mensagem centralizada na linha especificada com a cor especificada.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
______________________ CENTRALIZA ( _____________________ );

void main (void)
{
 clrscr();
```

```
 CENTRALIZA(12, "Jogo de Damas", BLUE);
 getch();
}
```

- ✓ Escreva em C a função **HIPOTENUSA**. A função recebe o valor da **base** e da **área** de um triângulo. A função deve devolver na variável altura a **altura do triângulo**. A função deve retornar também o **valor da hipotenusa**.

```
#include <stdio.h>
#include <conio.h>
____________________ HIPOTENUSA ( ________________ );

void main (void)
{
 float base, area, h, altura

 clrscr();
 printf("Base:  ");
 scanf("%f", &base);
 printf("Área do Círculo: ");
 scanf("%f", &area);
 h = HIPOTENUSA(base, area, &altura);
 printf("Hipotenusa: %.1f\n", h);
 printf("Altura: %.1f\n", altura);
 getch();
}
```

$$\text{área} = \frac{\text{base} \cdot \text{altura}}{2}$$

$$\text{hipotenusa}^2 = \text{base}^2 + \text{altura}^2$$

- ✓ Escreva em C a função **VERIFICA**. A função recebe um número inteiro (**n**). A função deve devolver na variável resto o **resto inteiro da divisão**. A função deve retornar se o número (**n**) é par (**1**) ou ímpar (**0**).

```
#include <stdio.h>
#include <conio.h>
_____________________ VERIFICA ( _____________________ );

void main (void)
{
 int n, resto;

 clrscr();
 printf("Número: ");
 scanf("%d", &n);
 if (VERIFICA(n, &resto))
        printf("Par\n");
 else
        printf("Ímpar\n");
 printf("Resto Inteiro da Divisão: %d\n", resto);
 getch();
}
```

- ✓ Escreva as seguintes funções: **STRCPY** (copia strings) e **STRCAT** (concatena strings)

```
#include <stdio.h>
#include <conio.h>
________________________ STRCPY (__________________________________________)
________________________ STRCAT (__________________________________________)

void main (void)
{
 char s[] = “Liber”, r[] = “dade”,t[10];

 clrscr();
 STRCPY(t, s);              /* função copia s para t → “Liber” */
 STRCAT(t, r);              /* insere r no final de t → “Liberdade” */
 printf(“%s\n”, t);         /* t → “liberdade” */
 getch();
}
```

14. Ponteiros

Um ponteiro é uma variável que contém um endereço de memória. O endereço pode ser a localização de uma ou mais variáveis na memória RAM ou qualquer outro endereço da memória RAM, como por exemplo, a memória da tela.

Figura 6: Representação de um ponteiro na memória

Exemplo:

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 char x = 65;
 char *p;

 p = &x;                              /* p recebe o endereço de x (&x) */
 printf("Valor de x...: %d\n", *p);   /* *p valor é 65 */
 printf("Caracter ....: %c\n", *p);   /* caracter 'A' */
printf("Endereço de x: %p\n", p);     /* endereço de x */
printf("Endereço de p: %p\n",&p);     /* endereço do ponteiro p */
getch();
}
```

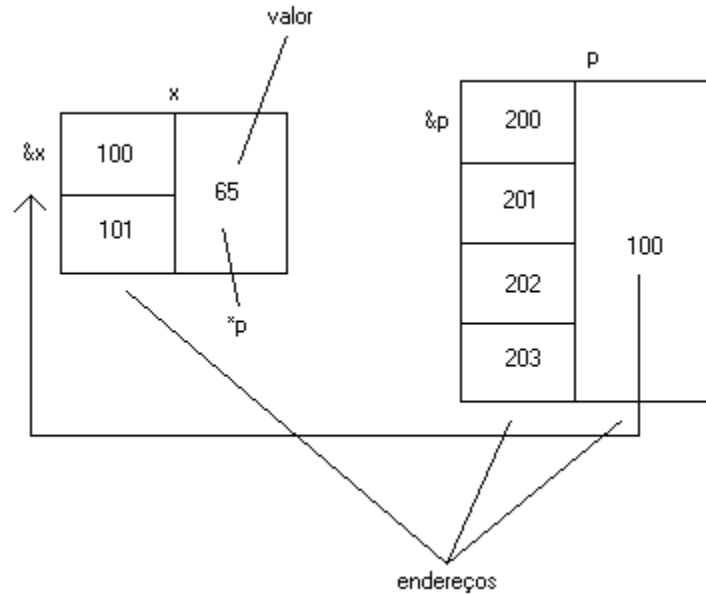



Figura 7: Endereçamento de um ponteiro

Resumo:

x ← 65 &x ← 100 p ← &x ←100 *p ← 65 &p ← 200

Um ponteiro pode ocupar 2 ou 4 bytes na memória RAM. No modelo de memória (model) **small** um ponteiro ocupa 2 bytes (tipo **near** - "perto") e no modelo de memória **large** ocupa 4 bytes (tipo **far** - "longe").

14.1 - Variáveis ponteiros

Definição:

tipo_dado_base *nome_do_ponteiro;

tipo_dado_base: qualquer tipo básico válido em C.
nome_da_ponteiro: nome da variável que representa o ponteiro.

O tipo de dados do ponteiro define para que tipos de variáveis o ponteiro pode apontar e qual o tamanho ocupado na memória por estas variáveis.

14.2 - Operadores de ponteiros

& → endereço de memória do operando.
***** → conteúdo do endereço apontado pelo ponteiro.

Exemplo: ponteiro = &variável;

Logo: variável = *ponteiro;

Observação: Como C sabe quantos bytes **copiar** para a variável apontada pelo ponteiro?

Resposta: O tipo_dado_base do ponteiro determina o tipo de dado que o ponteiro está apontando.

14.3 - Expressões com ponteiros

14.3.1 - Atribuições com ponteiros

Pode-se atribuir o valor (endereço) de um ponteiro para outro ponteiro.

Programa exemplo (42): O programa mostra atribuições utilizando dois ponteiros.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int x = 7;
 int *p1, *p2;

 clrscr();
 p1 = &x;                    /* p1 recebe o endereço de x */
 p2 = p1;                    /* p2 recebe o endereço de x */
 printf("%p\n", &x);         /* impresso endereço de x */
 printf("%p\n", p1);         /* impresso endereço de x */
 printf("%p\n", p2);         /* impresso endereço de x */
 printf("%d", *p1);          /* impresso valor de x (conteúdo de p1) */
 printf("%c", *p1);          /* impresso caracter 7 (Bell - conteúdo de p1) */
 getch();
}
```

14.3.2 - Aritmética com ponteiros

14.3.2.1 - Incremento (++)

Faz com que o ponteiro aponte para a localização de memória do próximo elemento de seu tipo_dado_base.

Exemplo:

```
p++;        /* ponteiro aponta para o próximo elemento */
(*p)++;  /* conteúdo do ponteiro é incrementado */
```

Programa exemplo (43): O programa mostra um ponteiro apontando para os elementos de um vetor. Isto é feito através da aritmética de ponteiros.

```
/* Compile o programa utilizando o modelo de memória large */

#include <stdio.h>
#include <conio.h>

void main (void)
{
 char x [10] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
 char *p;
 int i;

 clrscr();
 p = &x [0];
 for (i = 0; i <= 9; i++)
     {
      printf("Endereço: %p | Valor: x [%d] = %c\n", p, i, *p);
      p++;
     }
 getch();
}
```

Resultado na tela:

```
Endereço: 87D8:0FB4 | Valor: x[0] = A
Endereço: 87D8:0FB5 | Valor: x[1] = B
Endereço: 87D8:0FB6 | Valor: x[2] = C
Endereço: 87D8:0FB7 | Valor: x[3] = D
Endereço: 87D8:0FB8 | Valor: x[4] = E
Endereço: 87D8:0FB9 | Valor: x[5] = F
Endereço: 87D8:0FBA | Valor: x[6] = G
Endereço: 87D8:0FBB | Valor: x[7] = H
Endereço: 87D8:0FBC | Valor: x[8] = I
Endereço: 87D8:0FBD | Valor: x[9] = J
```

Programa exemplo (44): O programa mostra um ponteiro apontando para os elementos de um vetor. Isto é feito através da aritmética de ponteiros.

```
/* Compile o programa utilizando o modelo de memória large */

#include <stdio.h>
#include <conio.h>

void main(void)
{
 int x [10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 int i,*p;

 clrscr();
 p = &x[0];
 for (i = 0; i <= 9; i++)
     {
      printf("Endereço: %p | Valor: x [%d] = %d\n", p, i, *p);
      p++;
     }
```

```
 getch();
}
```

Resultado na Tela:

```
Endereço: 87D8:0FAA | Valor: x[0] = 0
Endereço: 87D8:0FAC | Valor: x[1] = 1
Endereço: 87D8:0FAE | Valor: x[2] = 2
Endereço: 87D8:0FB0 | Valor: x[3] = 3
Endereço: 87D8:0FB2 | Valor: x[4] = 4
Endereço: 87D8:0FB4 | Valor: x[5] = 5
Endereço: 87D8:0FB6 | Valor: x[6] = 6
Endereço: 87D8:0FB8 | Valor: x[7] = 7
Endereço: 87D8:0FBA | Valor: x[8] = 8
Endereço: 87D8:0FBC | Valor: x[9] = 9
```

14.3.2.2 - Decremento (--)

Faz com que o ponteiro aponte para a localização do elemento anterior.

Exemplo:

```
p--;            /* ponteiro aponta para o elemento anterior */
(*p)++;         /* conteúdo do ponteiro é incrementado */
```

14.3.3 - Soma (+) e subtração (-)

Exemplo: p = p + 9;

Faz com que o ponteiro **p** aponte para o nono (9º) elemento do tipo_dado_base, após aquele que o ponteiro estava apontando no momento.

Observação: Somente pode-se somar e subtrair números inteiros a ponteiros.

14.3.4 - Comparação de ponteiros

É possível comparar dois ponteiros através utilizando os operadores relacionais.

Exemplo:

```
if (p < q)
   printf("Endereço de p é menor do que q\n");
```

14.4 - Ponteiros e vetores

Em C, o nome de um vetor sem índice é o endereço do primeiro elementos da matriz.

Exemplo:

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int x[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
 int *p;

 p = &x[0];                    /* é igual a p = x */
   :
}
```

Para acessar o quinto (5º) elemento podemos escrever:

```
x[4];          ou          *(p+4);
```

Observação: Aritmética de ponteiros pode ser mais rápida que a indexação de vetores e matrizes.

14.4.1 - Indexando um ponteiro

Exemplo:

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int x[5] = { 0, 1, 2, 3, 4}
 int *p,t;

 p = x;                         /* igual a p = &x[0] */
 for (t = 0; t <= 4; t++)
     printf("%d\n", p[t]);
 getch();
}
```

14.4.2 - Ponteiros e strings

Como o nome de um vetor sem índice é um ponteiro que aponta para o primeiro elemento do vetor, quando utiliza-se funções que recebem *strings* como parâmetros, estas recebem apenas um ponteiro para a *string* e não o valor real da string em si, ou seja, sempre a passagem de parâmetro de uma string é por referência.

Programa exemplo (45):

```
#include <stdio.h>
#include <conio.h>
```

```
int compare (char *s1, char *s2);

void main (void)
{
 char s1[41], s2[41];

 clrscr();
 printf("String 1: ");
 gets(s1);
 printf("String 2: ");
 gets(s2);
 if (compare (s1,s2) == 0)                          /* s1 igual s2 == 0 */
    printf("s1 é igual a s2\n");
 else
    printf("s1 é diferente de s2\n");
 getch();
}

int compare (char *s1, char *s2)
{
 while (*s1)                    /* quando o *s1 for igual '\0' termina o laço */
      if (*s1-*s2)
        return (*s1-*s2);
      else
         {
          s1++;
          s2++;
         }
 return ('\0');
}
```

14.4.3 - Obtendo o endereço de um elemento de um vetor

```
p = &x[2];
```

14.4.4. Vetores de ponteiros

Declaração de um vetor de ponteiros de inteiros de tamanho 10.

```
int *x[10];
```

Atribuição do endereço de uma variável ao terceiro elemento da matriz de ponteiros:

x[2] = &variável;   **Logo**: Para obter o valor da variável, utiliza-se: *x[2];

14.5 - Ponteiros para ponteiros

Uma matriz de ponteiros é igual a apontar ponteiros para ponteiros. Um ponteiro para um ponteiro é uma forma de indireção múltipla.

Programa exemplo (46): O programa utiliza uma variável ponteiro para ponteiro.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int x;
 int *p;
 int **t;

 clrscr();
 x = 10;
 p = &x;
 t = &p;
 printf("%d", **t);                    /* imprime o valor de x */
 getch();
}
```

14.6 - Inicialização de ponteiros

Após declarar um ponteiro, e antes de haver uma atribuição de um valor inicial, o ponteiro terá um endereço desconhecido (lixo), ou seja, nunca se deve utilizar um ponteiro antes de atribuir-lhe um valor (endereço).

Podemos inicializar um ponteiro para caracteres (*string*) da seguinte forma:

Exemplo:

```
void main (void)
{
 char *p = "Paola";
 char *erro = "Falta de memória\n";
```

14.7 - Alocação dinâmica de memória

Permite alocar e liberar uma área de memória para variáveis durante a execução de um programa. Qualquer outro tipo de variável, que não seja um ponteiro, deve ser alocada estaticamente, ou seja, a variável não pode ocupar mais espaço do que foi declarado.

Exemplo:

```
int x [10];
/* um inteiro ocupa 2 bytes */
/* logo 10 inteiros ocupam 20 bytes na memória RAM durante toda a execução do programa */
```

14.7.1 – malloc

A função **malloc** (**m**emory **alloc**ation) permite alocar uma porção de memória dinamicamente.

Sintaxe: void **malloc** (int número_de_bytes);
Prototype: alloc.h e stdlib.h

A função **malloc** reserva espaço na memória RAM (**aloca**) a quantidade de bytes especificada pelo parâmetro da função (número_de_bytes), devolvendo um ponteiro do tipo **void*** apontando para o primeiro byte alocado.

O tipo **void*** representa um ponteiro sem tipo que deve ser "tipado" de acordo com o tipo de dado base do ponteiro. Isto pode ser feito utilizando-se um **cast**.

Quando não houver memória suficiente para alocar o ponteiro, a função **malloc** devolve um ponteiro nulo (**NULL**).

Exemplo:

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>

void main (void)
{
 int *p, n;

 clrscr();
 printf("Quantidade de elementos: ");
 scanf("%d",&n);
 p = (int *) malloc( n * sizeof (int) );
 if (p == NULL)                                   /* ou   if (!p) */
    printf("ERRO FATAL: Falta de memória\n");
 else
     {
      printf("Ok, memória alocada\n");
      printf("Endereço reservado: %p\n", p);
      printf("Quantidade de elementos alocados: %d\n", n);
      printf("Quantidade de bytes alocados: %d\n", n * sizeof(int));
     }
}
```

14.7.2 – free

A função **free** (livre) permite liberar a porção de memória alocada pela função **malloc**.

Sintaxe: void **free** (void *p);
Prototype: alloc.h e stdlib.h

A função **free** libera a área de memória alocada pela função **malloc**.

Programa exemplo (47): O ponteiro **p** aponta para uma região da memória com 80 bytes reservados (alocados), ou seja, 40 inteiros.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>

void main(void)
{
 int *p, t;

 clrscr();
 p = (int *) malloc( 40 * sizeof (int) );
 if (!p)
    printf("Erro Fatal: Falta de memória\n");
 else
    {
     for (t = 0; t <= 39;++t)
         *(p+t) = t;
     for (t = 0; t <= 39; ++t)
          printf("%d ", *(p+t));
     free(p);
    }
}
```

15. Entrada e saída em disco

Existem dois sistemas de arquivo definidos em C. O primeiro, definido pelo padrão ANSI e UNIX chamado sistema de arquivo **bufferizado** (formatado ou de alto nível) e o segundo, é definido apenas pelo UNIX chamado sistema de arquivos **não-bufferizado** (não-formatado ou de baixo nível).

15.1 - Fila de bytes (stream)

A fila de bytes é um dispositivo lógico de entrada ou saída de dados, independente do dispositivo real. Um arquivo (dispositivo externo) deve ser associado a uma fila de bytes.

C utiliza 5 filas de texto pré-definidas, são elas:

Tabela 18: Filas de texto

Fila	Função da fila
stdin	Entrada padrão (standard input)
stdout	Saída padrão (standard output)
stderr	Erro padrão (standard error)
stdprn	Saída para impressora (standard printer)
stdaux	Saída auxiliar (standard auxiliary)

15.1.1 - Filas de texto

Uma fila de texto é uma seqüência de caracteres organizada em linhas. Cada linha é finalizada por um caracter **'\n'**.

Pode ocorrer certas traduções de caracteres, tal como: **'\n'** ser convertida em CR (13) + LF (10). Dessa forma, pode não haver correspondência de 1 para 1 entre os caracteres que o computador escreve (lê) e aqueles no dispositivo externo.

CR – **C**arriage **R**eturn (retorno do carro - cursor)
LF – **L**ine **F**eed (avanço de linha)

CR + LF = <**enter**>

Exemplo de um arquivo texto:

ABC\n
ABCDEF\n
\n
ABCDEFGH\n
EOF

Observações: EOF – **E**nd **O**f **F**ile (fim do arquivo).
Todo arquivo texto possui um nome.

Nome do arquivo: 8 caracteres (nome) + 3 caracteres (extensão).

Exemplos: turboc.txt, turboc.tex, turboc.doc

15.1.2 - Filas binárias

Uma fila binária possui correspondência unívoca (1 para 1) com os bytes do dispositivo externo. Portanto nenhuma tradução de caracteres ocorrerá.

Um arquivo binário pode conter tipos de dados diferentes, como por exemplo: **char**, **int**, **float**, vetores, ponteiros, strings, etc....

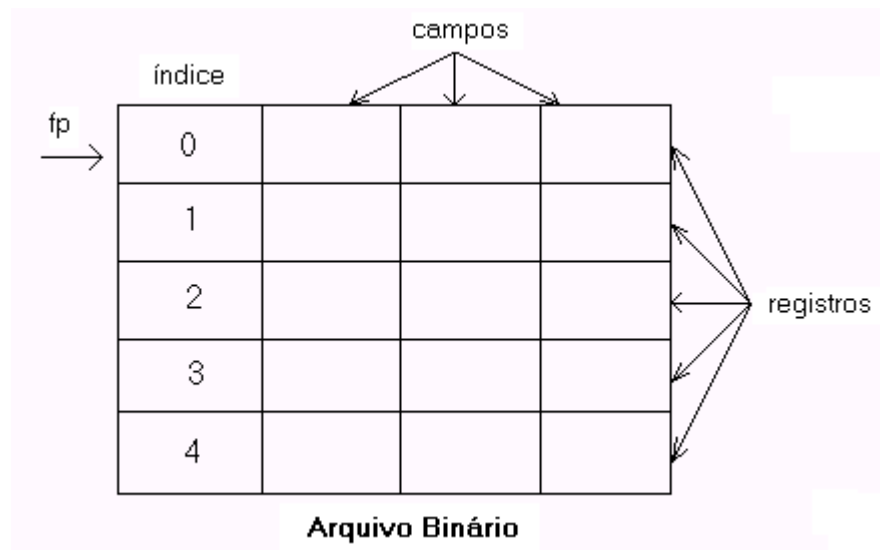



Figura 8: Representação de um arquivo

15.2 - Sistema de arquivo bufferizado

A ligação comum que mantém unido o sistema de entrada e saída bufferizado é um ponteiro que aponta para o arquivo. O ponteiro do arquivo identifica um determinado arquivo em disco e é usado pela fila associada a ele para direcionar cada uma das funções de entrada e saída bufferizada para o lugar em que elas operam. Um ponteiro de arquivo é uma variável de ponteiro do tipo **FILE** *. A palavra **FILE** é pré-definida em "**stdio.h**".

Exemplo:

```
#include <stdio.h>

void main(void)
{
 FILE *fp;
```

15.2.1 - fopen

A função ***fopen*** (file open) permite abrir um arquivo ligando-o a uma fila de bytes.

Sintaxe: FILE ***fopen** (char *nome_arquivo, char *modo);
Prototype: stdio.h

nome_arquivo: Deve ser uma string que contenha: "**drive:\path\nome_do_arquivo**"
modo: É uma string que contém as características desejadas (veja tabela abaixo).

Observação: **t** (arquivo texto) e **b** (arquivo binário)

Tabela 19: Modo de abertura de arquivos

Modo	Significado
“r”	***Abre*** um arquivo-texto para leitura
“w”	***Cria*** um arquivo-texto para gravação
“a”	***Anexa*** a um arquivo-texto
“rb”	Abre um arquivo binário para leitura
“”wb”	Cria um arquivo binário para gravação
“ab”	Anexa a um arquivo binário
“r+”	Abre um arquivo-texto para leitura/gravação
“w+”	Cria um arquivo-texto para leitura/gravação
“a+”	Abre ou cria um arquivo-texto para leitura/gravação
“r+b”	Abre um arquivo binário para leitura/gravação
“w+b”	Cria um arquivo binário para leitura/gravação
“a+b”	Abre um arquivo binário para leitura/gravação
“rt”	Abre um arquivo texto para leitura
“wt”	Cria um arquivo texto para gravação
“at”	Anexa a um arquivo texto
“r+t”	Abre um arquivo-texto para leitura/gravação
“w+t”	Cria um arquivo-texto para leitura/gravação
“a+t”	Abre ou cria um arquivo-texto para leitura/gravação

Observação: Se ocorrer ERRO na abertura de um arquivo, **fopen** devolverá um ponteiro nulo, ou seja, se (fp == NULL) erro.

Exemplos:

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 FILE *fp;

 clrscr();
 if ((fp = fopen("teste.dat","r")) == NULL)
      printf("Erro Fatal: Impossível abrir o arquivo\n");
 else
      {
       printf("Ok, arquivo aberto\n");
       ...
```

ou

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 FILE *fp;

 clrscr();
 fp = fopen("a:\fontes\teste.dat","w");
 if (fp == NULL)
     printf("Erro Fatal: Impossível criar o arquivo\n”);
```

```
else
     {
      printf("Ok, arquivo criado com sucesso\n");
        ...
```

15.2.2 – putc

A função **putc** é utilizada para gravar caracteres em um arquivo.

Sintaxe: int **putc** (int ch, FILE *fp);
Prototype: stdio.h

ch é o caracter a ser gravado
fp é o ponteiro do arquivo aberto pela função fopen()

Observação: Se uma gravação for bem sucedida putc() devolverá o caracter gravado, caso contrário devolverá um EOF.

EOF - End of File (Fim de Arquivo)

15.2.3 – getc

A função **getc** é utilizada para ler caracteres de um arquivo.

Sintaxe: int **getc** (FILE *fp);
Prototype: stdio.h

fp é o ponteiro do arquivo aberto por fopen()

15.2.4 – feof

A função **feof** (**f**ile **e**nd **o**f **f**ile) determina se o fim de arquivo foi encontrado. Devolve 0 se não chegou ao fim do arquivo.

Sintaxe: int **feof** (FILE *fp);
Prototype: stdio.h

Programa exemplo (48): O programa lista na tela todas as linhas de um arquivo texto, numerando-as.

Observação: Este programa deve ser compilado e executado pelo sistema operacional (".EXE") da seguinte forma:

A:\>LISTA lista.c <enter>

Funcionamento: Todas as linhas do programa fonte "lista.c" são exibidos na tela com as linhas numeradas. Ao preencher toda a tela ocorre uma pausa. A cada

parada é informado o nome do arquivo texto que está sendo listado e o número de bytes listados.

```
#include <stdio.h>
#include <conio.h>

void main (int argc, char *argv[])
{
 FILE *fp;
 int i, n;
 unsigned int v;
 char ch;

 if (argc != 2)
     printf("Sintaxe: LISTA <nome do arquivo>");
 else
     if ((fp = fopen(argv[1],"r")) == NULL)
         printf("ERRO: Arquivo [%s] não EXISTE\n");
     else
         {
          i = n = v = 1;
          printf("%d: ", i);
          do {
                ch = getc(fp);
                v++;
                if (ch == '\n')
                  {
                    i++;
                    if (n < 23)
                         n++;
                    else
                         {
                           n = 1;
                            printf("\n-------------------------------------------------------------------------------");
                            printf("[%s] Bytes: %d \t<ENTER>",argv[1],v);
                            getch();
                          }
                  printf("\n%d: ", i);
                  }
              else
                   printf("%c", ch);
             } while (c != EOF);
         printf("\n--------------------------------------------------------------------------------");
         printf("[%s]  Bytes: %d  ", argv[1],v);
         fclose(fp);
       }
}
```

15.2.5 – fclose

A função **fclose** (**f**ile **close**) é utilizada para fechar um arquivo aberto. Antes de fechar o arquivo os dados (que ainda estavam no **buffer**) são gravados.

Sintaxe: int **fclose** (FILE *fp);

Prototype: stdio.h

Observação: Retorna 0 se a operação foi bem sucedida.

15.2.6 - rewind

A função **rewind** estabelece o localizador de posição do arquivo para o início do arquivo especificado.

Sintaxe: void **rewind** (FILE *fp);
Prototype: stdio.h

15.2.7 – getw e putw

As funções **getw** e **putw** são utilizadas para ler e gravar, respectivamente, inteiros em um arquivo.

Função: Leitura de inteiros (**getw**)
Sintaxe: int **getw** (FILE *fp);
Prototype: stdio.h

Função: Gravação de inteiros (**putw**)
Sintaxe: int **putw** (int x, FILE *fp);
Prototype: stdio.h

15.2.8 – fgets e fputs

As funções **fgets** e **fputs** são utilizadas para ler e gravar strings.

Função: Leitura de strings (**fgets**)
Sintaxe: char ***fgets** (char *str, int comprimento, FILE *fp);
Prototype: stdio.h

Função: Gravação de strings (**fputs**)
Sintaxe: char ***fputs** (char *str, FILE *fp);
Prototype: stdio.h

Observação: A função **fgets** lê uma string do arquivo especificado até que leia ou um '**\n**' ou (comprimento - 1) caracteres.

15.2.9 – fread e fwrite

As funções **fread** e **fwrite** são utilizadas para ler e gravar blocos de dados, normalmente uma **struct**.

Função: Leitura de blocos (**fread**)
Sintaxe: int **fread** (void *buffer, int num_bytes, int cont, FILE *fp);

Prototype: stdio.h

Função: Gravação de blocos (**fwrite**)
Sintaxe: int **fwrite** (void *buffer, int num_bytes, int cont, FILE *fp);
Prototype: stdio.h

buffer: É um ponteiro para a região da memória ou o endereço de uma variável que receberá os dados lidos do arquivo pela função **fread** ou que será gravada no arquivo pela função **fwrite**.
num_bytes: Especifica a quantidade de bytes a serem lidos ou gravados.
cont: Determina quantos blocos (cada um com comprimento de num_bytes) serão lidos ou gravados.
fp: É o ponteiro para o arquivo.

15.2.10 – fseek

A função **fseek** é utilizada para ajustar o localizador de posição do arquivo, ou seja, permite selecionar a posição para efetuar operações de leitura e gravação aleatórias.

Sintaxe: int **fseek** (FILE *fp, long int num_bytes, int origem);
Prototype: stdio.h

num_bytes: É o número de bytes desde origem até chegar a posição desejada.

Tabela 20: Origem em arquivos

Origem	**Identificador**
Início do arquivo	SEEK_SET
Posição corrente	SEEK_CUR
Fim do arquivo	SEEK_END

15.2.11 – fprintf e fscanf

As funções **fprintf** e **fscanf** se comportam exatamente como **printf** e **scanf**, exceto pelo fato de que elas operam com arquivos em disco.

Função: Gravação de dados formatados (**fprintf**)
Sintaxe: int **fprintf** (FILE *fp, char *formato, lista argumentos);
Prototype: stdio.h

Função: Leitura de dados formatados (**fscanf**)
Sintaxe: int **fscanf** (FILE *fp, char *formato, lista argumentos);
Prototype: stdio.h

15.2.12 – remove

A função **remove** apaga do disco o arquivo especificado.

Sintaxe: int **remove** (char *nome_arquivo);
Prototype: stdio.h

Exemplos:

Abaixo, são listados três programas: **cria.c**, **lista.c**, **consulta.c,** os quais possuem como registro, uma palavra com no máximo 40 caracteres.

Programa exemplo (49): O programa permite criar um arquivo de palavras permitindo a gravação destas palavras.

```
/* cria.c */

#include <stdio.h>
#include <conio.h>
#include <string.h>

void main (void)
{
 FILE *fp;
 char reg[40];
 char nome_do_arquivo[14];
 unsigned int n;
 char ch;

 clrscr();
 printf("Nome do arquivo: ");
 scanf("%s", nome_do_arquivo);
 if ((fp = fopen(nome_do_arquivo,"r+b")) != NULL)
     {
      printf("ERRO: Arquivo já existe\n");
      getch();
      fclose(fp);
     }
else
    if ((fp = fopen(nome_do_arquivo,"w+b")) == NULL)
       {
        printf("Erro Fatal: Problema no disco\n");
        getch();
       }
    else
         {
          n = 0;
          do {
                clrscr();
                printf("%d: Palavra: ", n);
                scanf("%s", reg);                          /* scanf não aceita espaços */
                fwrite(reg, sizeof(reg), 1, fp);           /* grava o registro */
                n++;
                printf("Continua [S]im ou [N]ão ?");
                do {
                      ch = getch();
                     } while (!strchr("SsNn", ch));
```

```
        } while (strchr("Ss", ch));
      fclose(fp);
     }
}
```

Programa exemplo (50): O programa permite abrir o arquivo de palavras e exibe-os na tela.

```
/* lista.c */

#include <stdio.h>
#include <conio.h>

void main (void)
{
 FILE *fp;
 char reg[40];
 char nome_do_arquivo[14];
 unsigned int n;

 clrscr();
 printf("Nome do arquivo: ");
 scanf("%s", nome_do_arquivo);
 if ((fp = fopen(nome_do_arquivo,"rb")) == NULL)
   {
    printf("ERRO: Arquivo não EXISTE\n");
    getch();
   }
 else
    {
     n = 0;
     fread(reg, sizeof(reg), 1, fp);
     while (!feof(fp))
          {
           printf("%d: Palavra: %s\n", n, reg);
           n++;
           getch();
           fread(reg, sizeof(reg), 1, fp);
          }
    fclose(fp);
   }
}
```

Programa exemplo (51): O programa permite consultar o arquivo de palavras. Para tanto é solicitado, ao usuário, o número do registro para ser calculado a posição deste registro no arquivo. Logo após o registro é exibido na tela.

```
/* consulta.c */

#include <stdio.h>
#include <conio.h>
#include <string.h>

void main (void)
{
 FILE *fp;
 char reg[40];
```

```
 char nome_do_arquivo[14];
 unsigned int n,ok;
 long int posicao;
 char ch;

 clrscr();
 printf("Nome do arquivo: ");
 scanf("%s", nome_do_arquivo);
 if ((fp = fopen(nome_do_arquivo,"r+b")) == NULL)
   {
    printf("ERRO: Arquivo não EXISTE\n");
    getch();
   }
 else
    {
     do {
           clrscr();
           printf("Número do registro: ");
           scanf("%d", &n);
           posicao = n * sizeof(reg);
           fseek(fp, posicao, SEEK_SET);
           ok = fread(reg, sizeof(reg), 1, fp);
           if (ok)
              printf("%d: Palavra: %s\n", n, reg);
           else
               printf("ERRO: Registro NÃO existe\n");
            printf("Continua [S/N] ? ");
            do {
                 ch = getch();
                } while (!strchr("SsNn",ch));
           } while (strchr("Ss",ch));
    fclose(fp);
   }
}
```

15.3 - Sistema de arquivo não bufferizado

Ao contrário do sistema de entrada e saída de alto nível (sistema bufferizado), o sistema de baixo nível (sistema não-bufferizado) não utiliza ponteiros de arquivo do tipo **FILE**, mas sim descritores de arquivos chamados **handles** do tipo **int**, ou seja, uma variável inteira.

15.3.1 – open, creat e close

A função **open** permite abrir/criar um arquivo em disco.

Função: Abre um arquivo (**open**)
Sintaxe: int **open** (char *nomearquivo, int acesso);
Prototype: io.h

Tabela 21: Tipos de acessos em arquivos

Acesso	Efeito
O_RDONLY	Apenas leitura
O_WRONLY	Apenas gravação
O_RDWR	Leitura e gravação
O_CREAT	Cria e abre
O_TRUNC	Abre com "truncation"
O_EXCL	Abre exclusiva
O_APPEND	Abre para incluir no fim
O_TEXT	Translação CR para LF
O_BINARY	Sem translação

Observação: Se a função **open** obtiver sucesso (na abertura do arquivo), será devolvido um inteiro positivo. Um valor de retorno -1 (EOF) significa que **open** não pode abrir o arquivo. (O acesso se relaciona ao Ambiente UNIX)

A função **creat** cria um arquivo para operações de leitura e gravação.

Função: Cria um arquivo (**creat**)
Sintaxe: int **creat** (const char *nomearquivo, int modo);
Prototype: io.h

Tabela 22: Modos de acessos em arquivos

Modo	Efeito
S_IFMT	Arquivo tipo máscara
S_IFDIR	Diretório
S_IFIFO	FIFO especial
S_IFCHR	Caracter especial
S_IFBLK	Bloco especial
S_IFREG	Arquivoregular
S_IREAD	Proprietário pode ler
S_IWRITE	Proprietário pode escrever
S_IEXEC	Proprietário pode executar

A função **close** fecha um arquivo. Devolve -1 (EOF), se não for capaz de fechar o arquivo.

Função: Fecha um arquivo (**close**)
Sintaxe: int **close** (int fd);
Prototype: io.h

15.3.2 – write e read

A função **write** grava tantos caracteres quantos forem o tamanho do buffer apontado pelo ponteiro **buf** para o arquivo em disco especificado por **fd**.

Função: Grava caracteres no arquivo (**write**)
Sintaxe: int **write** (int fd, void *buf, int tamanho);
Prototype: io.h

A função **read** copia os dados lidos no **buffer** apontado pelo ponteiro **buf**.

Função: Lê caracteres do arquivo (**read**)
Sintaxe: int **read** (int fd, void *buf, int tamanho);
Prototype: io.h

15.3.3 – unlink

A função **unlink** elimina um arquivo do disco (retorno: 0 sucesso ou 1 erro).

Sintaxe: int **unlink** (const char *nomearquivo);
Prototype: dos.h, io.h e stdio.h

15.3.4 – lseek

A função **lseek** devolve o número de bytes (num_bytes) se obtiver sucesso.

Sintaxe: long int **lseek** (int fd, long int num_bytes, int origem);
Prototype: io.h e stdio.h

fd: descritor do arquivo
num_bytes: número de bytes desde a origem até a nova posição
origem: SEEK_SET (início), SEEK_CUR (corrente), SEEK_END (fim)

15.3.5 – eof

A função **eof** (end of file) devolve: (**1**) **fim de arquivo**, (**0**) **não é fim de arquivo** ou (**-1**) **erro**.

Sintaxe: int int **eof** (int fd);
Prototype: io.h

15.3.6 – tell

A função **tell** devolve a posição corrente do descritor.

Sintaxe: long int **tell** (int fd);
Prototype: io.h

Programa exemplo (52): O programa permite criar um arquivo de palavras permitindo a gravação destas palavras.

```
/* cria.c */

#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <string.h>
```

```
#include <fcntl.h>
#include <sys\stat.h>

void main (void)
{
 int fd;
 char reg[40];
 char nome_do_arquivo[14];
 unsigned int n;
 char ch;

 clrscr();
 printf("Nome do arquivo: ");
 scanf("%s", nome_do_arquivo);
 if ((fd = open(nome_do_arquivo,O_RDONLY,"r")) == EOF)
    if ((fd = creat(nome_do_arquivo,S_IWRITE)) == EOF)
       {
         printf("Erro Fatal: Problema no disco\n");
         getch();
       }
    else
          {
            n = 0;
           do {
                  clrscr();
                  printf("%d: Palavra: ", n);
                  scanf("%s", reg);
                  write(fd, reg, sizeof(reg));
                  n++;
                  printf("Continua [S]im ou [N]ão ?");
                  do {
                        ch = getch();
                       } while (!strchr("SsNn",ch));
                } while (strchr("Ss",ch));
          close(fd);
         }
 else
     {
      printf("ERRO: Arquivo já EXISTE\n");
      getch();
      close(fd);
     }
}
```

Programa exemplo (53): O programa permite abrir o arquivo de palavras e exibe-os na tela.

```
/* lista.c */

#include <stdio.h>
#include <io.h>
#include <conio.h>
#include <fcntl.h>
#include <sys\stat.h>

void main (void)
{
```

```
 int fd;
 char reg[40];
 char nome_do_arquivo[14];
 unsigned int n;

 clrscr();
 printf("Nome do Arquivo: ");
 scanf("%s", nome_do_arquivo);
 if ((fd = open(nome_do_arquivo,O_RDONLY,"r")) == EOF)
    {
     printf("ERRO: Arquivo NÃO existe ...\n");
     getch();
    }
 else
      {
         n = 0;
         do {
               read(fd, reg, sizeof(reg));
               printf("%d: Palavra: %s\n", n, reg);
               n++;
               getch();
             } while (!eof(fd));
         close(fd);
        }
}
```

Programa exemplo (54): O programa permite alterar o arquivo de palavras. Para tanto é solicitado, ao usuário, o número do registro para ser calculado a posição deste registro no arquivo. Logo após o registro é exibido na tela e é solicitada a nova palavra.

```
/* altera.c */

#include <stdio.h>
#include <string.h>
#include <io.h>
#include <conio.h>
#include <fcntl.h>
#include <sys\stat.h>

void main (void)
{
 int fd;
 char reg[40];
 char nome_do_arquivo[14];
 unsigned int n;
 long int posicao;
 char ch;

 clrscr();
 printf("Nome do arquivo: ");
 scanf("%s", nome_do_arquivo);
 if ((fd = open(nome_do_arquivo,O_RDWR,"w")) == EOF)
    {
     printf("ERRO: Arquivo NÃO existe ...\n");
     getch();
    }
```

```
else
    {
     do {
          clrscr();
          printf("Registro: ");
          scanf("%d", &n);
          posicao = n * sizeof(reg);
          lseek(fd, posicao, SEEK_SET);
          if (read(fd, reg, sizeof(reg)))
            {
              printf("%d: Palavra: %s\n", n, reg);
              flushall();
              printf("NOVA PALAVRA: ");
              scanf("%s", reg);
              lseek(fd, posicao, SEEK_SET);
              write(fd, reg, sizeof(reg));
            }
        else
              printf("ERRO: Registro não existe\n");
        printf("Continua [S]im ou [N]ão ? ");
        do {
             ch = getch();
            } while (!strchr("SsNn",ch));
       } while (strchr("Ss",ch));
     close(fd);
    }
}
```

15.4 - Lista de exercícios (arquivos)

✓ Escreva um programa em C que recebe via teclado o **nome de um arquivo texto**. O programa deve imprimir na tela o **número de bytes** (caracteres) e o **número de linhas** do arquivo ou **ERRO: Arquivo não existe**.

Exemplo:

Nome do arquivo texto: **LISTA.C** <**enter**>
(**12345**) Bytes
(**44**) Linhas

ou

ERRO: Arquivo não existe

✓ Escreva um programa em C que recebe via teclado o **nome de um arquivo texto**. O programa deve permitir ao usuário consultar o arquivo através da entrada via teclado do **número da linha**. O programa deve imprimir a linha especificada ou **ERRO: Linha não existe**.

Exemplo:

Nome do arquivo texto: **LISTA.C** <**enter**>
Número de linha: **7** <**enter**>

7: int i, j, k;
Número de linha: **70** <**enter**>
ERRO: Linha não existe
Número de linha: **0** <**enter**>

ou

ERRO: Arquivo não existe

✓ Escreva um programa em C que recebe via teclado o **nome de dois arquivos texto** (origem e destino). O programa deve copiar o conteúdo do arquivo origem para o arquivo destino.

Exemplo:

Arquivo origem: **LISTA.C** <**enter**>
Arquivo destino: **LISTA.tmp** <**enter**>
(20345) Bytes copiados

✓ Escreva um programa em C que recebe via teclado o **nome do arquivo texto fonte** e o **nome do arquivo texto destino**. O programa deve converter o arquivo fonte para **maiúsculo** ou **minúsculo** (conforme escolha do usuário) gerando o arquivo texto destino.

Exemplo:

Arquivo fonte: **LISTA.C** <**enter**>
Arquivo destino: **LISTA.TMP** <**enter**>
[+] Maiúsculo ou [-] Minúsculo: **+**
(1234) Bytes convertidos

✓ Escreva um programa em C que recebe via teclado o **nome de um arquivo texto** e uma **palavra**. O programa deve imprimir todas as linhas que possuem esta palavra.

Exemplo:

Nome do arquivo texto: **PALAVRA.C** <**enter**>
Palavra: **if** <**enter**>
23: if (fp == NULL)
33: if (ch == '\n')
37: if (compara(linha,palavra))
41: if (ch != '\n')
59: if (linha[i] == palavra[0])
65: if (linha[k] != palavra[j])
69: if (achei)

✓ Escreva um programa em C que recebe via teclado o **nome de um arquivo texto**. O programa deve permitir ao usuário consultar o arquivo através da entrada via teclado do **número inicial** e **número final**. O programa deve imprimir desde a linha inicial até a linha final ou **ERRO: Linhas não existem**.

Exemplo:

Nome do arquivo texto: **LISTA.C** <**enter**>
Número inicial: **7** <**enter**>
Número final: **9** <**enter**>
7: int i, j, k;
8: char tecla;
9: long int bytes = 0;

ou

Número inicial: **70** <**enter**>
Número final: **90** <**enter**>
ERRO: Linhas não existem

✓ Escreva um programa em C (**grava.c**) que recebe via teclado o **nome de um arquivo binário**. O programa deve permitir ao usuário inserir nomes (máximo 30 caracteres) neste arquivo via teclado. O programa termina quando o usuário digitar <**enter**> na entrada do nome.

Exemplo:

Nome do arquivo binário: **NOMES.DAT** <**enter**>
Nome: **Beatriz** <**enter**>
Nome: **Eva** <**enter**>
Nome: **Debora** <**enter**>
Nome: **Carla** <**enter**>
Nome: **Fatima** <**enter**>
Nome: **Ana** <**enter**>
Nome: <**enter**>

✓ Escreva um programa em C (**ler.c**) que recebe via teclado o **nome de um arquivo binário**. O programa deve ler e imprimir na tela todos os nomes armazenados no arquivo pelo programa "**grava.c**".

Exemplo:

Nome do arquivo binário: **NOMES.DAT** <**enter**>
Nome: **Beatriz**
Nome: **Eva**
Nome: **Debora**
Nome: **Carla**
Nome: **Fatima**
Nome: **Ana**

✓ Escreva um programa em C (**sort.c**) que recebe via teclado o **nome de um arquivo binário**. O programa deve ler, ordenar e gravar novamente os nomes no mesmo arquivo.

Exemplo:

Nome do arquivo binário: **NOMES.DAT** <**enter**>
Ok, arquivo ordenado

Observação: Utilize o programa anterior (**ler.c**) para ver os nomes ordenados.

✓ Escreva um programa em C (**salva.c**) que recebe via teclado o **nome de um arquivo binário**. O programa deve permitir ao usuário inserir **nome** (máximo 30 caracteres), **idade** e **sexo** ([M]asculino ou [F]eminino) neste arquivo via teclado. O programa termina quando o usuário digitar <**enter**> na entrada do nome.

Exemplo:

Nome do arquivo binário: **DADOS.DAT** <**enter**>
Nome: **Paulo Roberto** <**enter**>
Idade: **41** <**enter**>
Sexo [M/F]: **m** <**enter**>
Nome: **Renato Luis** <**enter**>
Idade: **38** <**enter**>
Sexo [M/F]: **m** <**enter**>
Nome: **Ana Maria** <**enter**>
Idade: **44** <**enter**>
Sexo [M/F]: **f** <**enter**>
Nome: <**enter**>

✓ Escreva um programa em C (**list.c**) que recebe via teclado o **nome de um arquivo binário**. O programa deve ler e imprimir na tela todos os dados (nome, idade e sexo) armazenados no arquivo pelo programa "**salva.c**" (veja exemplo abaixo).

Exemplo:

Nome do arquivo binário: **DADOS.DAT** <**enter**>
Nome: **Paulo Roberto**
Idade: **41**
Sexo: **MASCULINO**
Nome: **Renato Luis**
Idade: **38**
Sexo: **MASCULINO**
Nome: **Ana Maria**
Idade: **44**
Sexo: **FEMININO**

✓ Escreva um programa em C (**conta.c**) que recebe via teclado o **nome de um arquivo binário**. O programa deve verificar a quantidade de homens e a quantidade de mulheres no aquivo criado pelo programa "**salva.c**".

Exemplo:

Nome do arquivo binário: **DADOS.DAT** <**enter**>
(2) Homens
(1) Mulheres

16. Tipos de dados definidos pelo programador

16.1 - Estruturas

Em C, uma estrutura (struct) é uma coleção de campos que fazem referência a uma variável, ou seja, uma variável possui um conjunto de valores (que podem ser de tipos iguais ou diferentes). Uma estrutura propicia uma maneira conveniente de manter-se juntas informações relacionadas.

```
struct nome_da_estrutura {
                              tipo nome_variável;
                              tipo nome_variável;
                                . . .
                            } lista_de_variáveis;
```

Na definição de uma estrutura pode-se omitir o **nome_da_estrutura** ou a **lista_de_variáveis** deste tipo, mas não ambos.

Exemplo:

```
struct reg_cliente {                        /* nome_da_estrutura */
                     char nome[30];
                     float salario;
                     int idade;
                   } cliente, fornecedor;          /* lista_de_variáveis */
```

16.1.1 - Referência aos elementos da estrutura

Para fazer referência aos elementos individuais de uma estrutura deve-se utilizar o operador "." (ponto), que as vezes é chamado de operador de seleção.

```
nome_variável_estrutura.nome_campo
```

Exemplo: cliente.idade = 18;

16.1.2 - Matrizes de estruturas

É possível criar matrizes de estruturas.

Exemplo: struct **reg_cliente cliente**[100];

Cria-se um **array** (vetor) para guardar 100 ocorrências de cliente.

Acesso ao primeiro elemento da estrutura:

```
strcpy(cliente[0].nome,"Paulo");
cliente[0].salario = 500.00;
cliente[0].idade = 41;
```

Acesso ao último elemento da estrutura:

```
strcpy(cliente[99].nome,"Renato");
```

```
cliente[99].salario = 1500.00;
cliente[99].idade = 37;
```

16.1.3 - Passando estruturas para funções

16.1.3.1 - Passando elementos da estrutura

Desta forma, é passado apenas uma cópia do valor para a função.

Chamada da função: função (cliente.idade);

16.1.3.2 - Passando estruturas inteiras

É possível passar uma estrutura (**struct**) inteira para uma função.

Observação: O tipo do argumento deve coincidir com o tipo do parâmetro.

Exemplo:

```
#include <stdio.h>
#include <conio.h>

struct TIPO {
                int a,b;
                char ch;
              };

void function (struct TIPO parm);

void main (void)
{
 struct TIPO arg;

 clrscr();
 arg.a = 1000;
 arg.b = 500;
 arg.ch = 'A';
 function (arg);              /* arg é o argumento */
}

void function (struct TIPO parm)            /* parm é o parâmetro tipo estrutura */
{
printf("%d", parm.a);
printf("%d", parm.b);
printf("%c", parm.ch);
}
```

16.1.4 - Ponteiros para estruturas

C permite que ponteiros apontem para estruturas da mesma maneira como permite ponteiros para qualquer outro tipo de variável.

Exemplo:

```
struct reg_cliente {
                     char nome[30];
                     float salario;
                     int idade;
                    } cliente;

struct reg_cliente *p;
```

Neste caso **p** é um ponteiro para a estrutura **reg_cliente**.

p = &cliente; /* o ponteiro **p** aponta para a variável **cliente** */

Para acessar um elemento de cliente através do ponteiro **p** deve-se escrever das seguintes formas:

(*p).idade = 21;   ou   p→idade = 21;

16.2 - Campos de bit

C possui um método inerente de acesso a um único **bit** em um **byte**. Este é baseado na estrutura. Um campo de bit é apenas um tipo especial de estrutura que define o comprimento em bits que cada elemento terá.

```
struct nome_tipo_estrutura {
                             tipo nome_1: comprimento;
                             tipo nome_2: comprimento;
                              ...
                             tipo nome_3: comprimento;
                            };
```

Observação: Um campo de **bit** deve ser do tipo **int**, **unsigned int** ou **signed int**.

Exemplo:

```
struct dispositivo {
                    unsigned ativo: 1;
                    unsigned pronto: 1;
                    unsigned erro: 1;
                   } codigo_disp;
```

Esta estrutura define três variáveis de um bit cada, os outros 5 bits ficam desocupados, pois **codigo_disp** ocupa 8 bits, ou seja, 1 byte.

16.3 - Union

Em C, uma **union** (união) é uma localização de memória que é utilizada por várias variáveis diferentes, que podem ser de tipos diferentes (**int** ou **char**).

Exemplo:

```
union u_type {
        int i;
        char ch;
       };
```

Neste caso a variável **i** e **ch** ocupam a mesma localização da memória.

16.4 - typedef

C permite que o programador defina explicitamente novos nomes de tipos de dados utilizando a palavra reservada **typedef.** Realmente não se cria uma nova classe de dados, mas sim, um novo nome para um tipo existente.

typedef tipo_existente novo_nome_de_tipo;

Exemplo: typedef float real;

real salario;

16.5 - Tipos de dados avançados

C permite que seja aplicado vários **modificadores de tipo** aos tipos de dados básicos.

modificador_tipo tipo_básico lista_variáveis;

16.5.1 - Modificadores de acesso

16.5.1.1 - O modificador const

Durante a execução, o programa não poderá alterar uma variável declarada com o modificador **const**, exceto dar a variável um valor inicial.

Exemplo: const float versao = 3.30;

16.5.1.2 - O modificador volatile

Utiliza-se o modificador **volatile** para dizer ao C que o valor de uma variável pode ser alterado sem uma especificação explícita do programa.

Exemplo: volatile int clock;

16.5.2 - Especificadores de classe de armazenamento

16.5.2.1 - O especificador auto

O especificador **auto** é utilizado para declarar variáveis locais. (Por default as variáveis locais são **auto**).

16.5.2.2 - O especificador extern

O especificador **extern** diz ao compilador que os tipos de dados e nomes de variáveis que se seguem já foram declarados em um outro lugar (por exemplo, outro programa fonte).

16.5.2.3 - O especificador static

As variáveis **static** são variáveis permanentes dentro de suas próprias funções. São diferentes das variáveis globais, porque não são conhecidas fora de suas funções, mas mantém seus valores entre as chamadas.

16.5.2.4. O especificador register

Se aplica apenas as variáveis **int** ou **char**. O especificador **register** faz com que o compilador mantenha o valor das variáveis declaradas dentro dos registradores da CPU, e não na memória, onde as variáveis normais são armazenadas normalmente. É próprio para variáveis de loop's (laços), pois torna o laço mais rápido.

Observação: Só pode ser utilizado em variáveis locais.

16.5.3 - Operadores avançados

16.5.3.1 - Operadores bit a bit

Se referem ao teste, ajuste ou troca de bits reais por um **byte** ou palavra (podem ser de tipos **char** ou **int**).

Tabela 23: Operadores bit a bit

Operador	Ação
&	*And*
|	*Or*
^	*Or* exclusivo (*XOr*)
~	Complemento de um (*Not*)
>>	Deslocar para a direita (*shift*)
<<	Deslocar para a esquerda (*shift*)

Deslocamento (shift): variável >> número de deslocamentos;

Exemplo:

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int a = 128, b, c;                    /* a = 128 → 10000000 */

 clrscr();
 b = a >> 1;                           /* b = 64 → 0100000 */
 c = b << 1;                           /* c = 128 → 1000000 */
 printf("a = %d\n",a);
 printf("b = %d\n",b);
 printf("c = %d\n",c);
 getch();
}
```

16.5.3.2 - O operador ?

Pode-se utilizar o operador **?** para substituir comandos **if ... else** que tenham a seguinte forma geral:

```
if (condição)
     comando_1;
else
     comando_2;
```

A principal restrição do operador **?** é que os comandos do **if** ... **else** devem ser comando simples não podendo serem comandos compostos.

Forma geral: condição ? comando_1: comando_2;

Exemplo:

```
#include <stdio.h>
#include <conio.h>

void main (void)
{
 int a, b;

 clrscr();
 printf("a = ");
 scanf("%d",&a);
 printf("b = ");
 scanf("%d",&b);
 a > b ? printf("Maior (a) -> %d\n",a) : printf("Maior (b) -> %d\n",b);
 getch();
}
```

16.5.3.3 - Formas abreviadas de C

C tem abreviações especiais que simplificam a codificação de um certo tipo de comando de atribuição.

Exemplo:

x = x + 10; → pode ser escrito como → x += 10;

Esta forma abreviada funcionará com todos os operadores aritméticos de C.

Forma normal: variável = variável **operador** constante;
x = x + 7;
Forma abreviada: variável **operador**= constante;
x += 7;

16.5.3.4 - O operador ,

Pode-se utilizar a vírgula para juntar várias expressões. O compilador sempre avalia o lado esquerdo da vírgula como **void**. Assim, a expressão do lado direito ficará sendo o valor de toda expressão separada por vírgula.

Exemplo: x = (y = 3, y + 1);

Primeiramente é atribuído o valor 3 a **y** e depois atribuído o valor 4 a **x**. Os parênteses são necessários, porque a vírgula tem precedência mais baixa que o operador de atribuição.

17. Gráficos

17.1 Placas gráficas

Dispositivos gráficos que permitem trabalhar com resoluções maiores do que o modo texto permite, ou seja, imprimir informações com mais detalhes. Uma placa gráfica possui uma memória interna que permite operar com grandes resoluções.

Resolução: Quantidade de pixels na horizontal **x** quantidade de pixels na vertical.

A tela do computador pode operar em modo texto ou modo gráfico. Em modo texto, normalmente, a tela é dividida em 80 colunas por 25 linhas. Em modo texto pode-se exibir apenas os 256 caracteres da tabela ASCII e poucas cores. A tela em modo gráfico é dividida, normalmente, em 1280 *pixels* na horizontal por

1024 *pixels* na vertical. Em modo gráfico é possível exibir qualquer tipo de objeto, imagem ou texto com milhares (ou milhões) de cores.

Pixel: Elemento gráfico (menor ponto representado em modo gráfico).

17.1.1 CGA

Resolução: 640 x 200 alta resolução (2 cores)
320 x 200 baixa resolução (4 cores de um *palette* de 16 cores)

Observação: *Palette* é o conjunto de cores disponíveis, cada *palette* (CGA) possui 4 cores, dentre as 16 cores disponíveis.

17.1.2 EGA

Resolução: 640 x 350 alta resolução (16 cores e 2 páginas gráficas)
640 x 200 baixa resolução (16 cores e 4 páginas gráficas)

17.1.3 VGA

Resolução: 640 x 480 alta resolução (16 cores e 1 página)
640 x 350 média resolução (16 cores e 2 páginas gráficas)
640 x 200 baixa resolução (16 cores e 4 páginas gráficas)

17.2 Coordenadas de tela

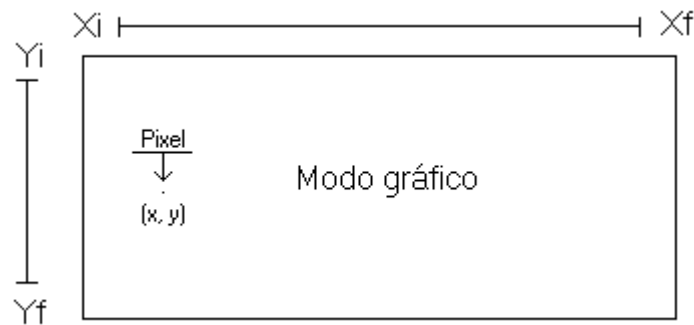



Figura 9: Coordenadas de tela em modo gráfico

17.2.1 CGA

Tabela 24: Coordenadas de tela (CGA)

Resolução	xi	yi	xf	Yf
320 x 200	0	0	319	199
640 x 200	0	0	639	199

17.2.2 EGA

Tabela 25: Coordenadas de tela (EGA)

Resolução	xi	yi	Xf	Yf
640 x 200	0	0	639	199
640 x 350	0	0	639	349

17.2.3 VGA

Tabela 26: Coordenadas de tela (VGA)

Resolução	xi	yi	Xf	Yf
640 x 200	0	0	639	199
640 x 350	0	0	639	349
640 x 480	0	0	639	479

17.3 Detecção e inicialização da tela gráfica

Para inicializar a tela em modo gráfico, ou seja, sair do modo texto, devem ser definidas duas variáveis inteiras: **placa** e **modo** (podem ter qualquer nome).

placa: define o tipo de placa gráfica que será utilizada, podendo assumir um dos seguintes tipos de placas:

Tabela 27: Tipos de placas

Número	Nome
0	DETECT
1	CGA
2	MCGA
3	EGA
4	EGA64
5	EGAMONO
6	IBM8514
7	HERCMONO
8	ATT400
9	VGA
10	PC3270

modo: define a resolução da placa gráfica (**alta**, **média** ou **baixa**), podendo ser:

Tabela 28: Modos das placas gráficas

Modo	Resolução	Número de cores
CGAC0	320 x 200	4 cores (palette 0)
CGAC1	320 x 200	4 cores (palette 1)
CGAC2	320 x 200	4 cores (palette 2)
CGAC3	320 x 200	4 cores (palette 3)
CGAHI	640 x 200	2 cores
MCGAC0	320 x 200	4 cores (palette 0)
MCGAC1	320 x 200	4 cores (palette 1)
MCGAC2	320 x 200	4 cores (palette 2)
MCGAC3	320 x 200	4 cores (palette 3)

MCGAMED	640 x 200	2 cores
MCGAHI	640 x 480	2 cores
EGALO	640 x 200	16 cores
EGAHI	640 x 350	16 cores
EGA64LO	640 x 200	16 cores
EGA64HI	640 x 350	4 cores
EGAMONOHI	640 x 350	2 cores
VGALO	640 x 200	16 cores
VGAMED	640 x 350	16 cores
VGAHI	640 x 480	16 cores
HERCMONOHI	720 x 348	2 cores

As variáveis **placa** e **modo** devem ser utilizadas na função **initgraph**. Esta função permite a inicialização do modo gráfico.

Sintaxe: void **initgraph** (int *placa, int *modo, char *path);
Prototype: graphics.h

ATENÇÃO: Se o caminho (**path)** for vazio ("") é sinal de que os arquivos relacionados as placas gráficas (*.bgi, por exemplo: **CGA.BGI**, **EGAVGA.BGI**) e os relativos as fontes de caracteres (*.chr, por exemplo: **GOTH.CHR**, **LITT.CHR**, **SANS.CHR** e **TRIP.CHR**) estão localizados no diretório corrente.

Observação: O diretório corrente pode ser consultado ou modificado no item de menu "**Change Dir"** do Turbo C 2.01.

Forma de utilizar: placa = DETECT;
initgraph (&placa, &modo, "C:\\TC\\BGI");

Os programas 55, 56 e 57 exemplificam algumas formas de inicializar a tela em modo gráfico.

Programa exemplo (55): O programa detecta a placa gráfica devido a instrução *placa* = ***DETECT;*** normalmente, como VGA. Sempre que o modo gráfico não for selecionado, o programa utiliza a maior resolução possível. (No caso de VGA será VGAHI). Um círculo é exibido na tela.

```
#include <graphics.h>
#include <conio.h>

void main (void)
{
 int placa, modo;
 int raio = 5;

 placa = DETECT;         /* vai haver uma detecção automática da placa gráfica */
 initgraph(&placa, &modo,"c:\\tc\\bgi");   /* "" se os arquivos *.bgi e *.chr estão diretório corrente */
 if (!graphresult())
   {
```

```
     do {
           circle(319, 99, raio);
           raio += 5;
         } while (raio <= 200);
     getch();
     closegraph();
    }
 else
     printf("Erro Fatal: Problema na inicialização da placa gráfica\n");
}
```

ou

Programa exemplo (56): O programa seleciona a placa gráfica como VGA e o modo gráfico como VGAHI. Um círculo é exibido na tela.

```
#include <graphics.h>
#include <conio.h>

void main (void)
{
 int placa, modo, raio;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 if (!graphresult())
    {
     raio = 5;
     do {
           circle(319, 239, raio);
           raio += 5;
         } while (raio <= 200);
     getch();
     closegraph();
    }
 else
       printf("Erro Fatal: Problema na inicialização da placa gráfica\n");
}
```

ou

Programa exemplo (57): O programa detecta a placa gráfica e desenha um círculo na na tela.

```
#include <graphics.h>
#include <conio.h>

void main (void)
{
 int placa, modo, raio = 5;
 int y[3] = {99, 174, 239};            /* posição Y (central) das placas */

 placa = DETECT;         /* detecta placa */
 initgraph(&placa, &modo, "");                /* inicializa modo gráfico */
 if (!graphresult())                          /* verifica se houve erro */
    {
```

```
    switch (placa)
      {
       case CGA: modo = CGAHI;             /* 640 x 200 */
                     break;
       case EGA: modo = EGAHI;           /* 640 x 350 */
                     break;
       case VGA: modo = VGAHI;           /* 640 x 480 */
                     break;
      }
    setgraphmode(modo);          /* seta modo */
    do {
             circle(319, y [placa - 1], raio);
             raio += 5;
            } while (raio <= 100);
    getch();
    closegraph();           /* saída do modo gráfico, retorna ao modo texto */
   }
 else
    printf("Erro Fatal: Problema na inicialização da placa gráfica\n");
}
```

17.4 putpixel

A função **putpixel** (desenha pontos) permite que um **pixel** seja exibido na tela gráfica.

Sintaxe: void **putpixel** (int x, int y, int cor_pixel);
Prototype: graphics.h

Programa exemplo (58): O programa exibe pontos aleatórios na tela.

```
#include <graphics.h>
#include <conio.h>       /* kbhit */
#include <stdlib.h>       /* random */

void main (void)
{
 int placa, modo;
 int x, y, cor;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 randomize();
 do {
      x = random (640);
      y = random (480);
      cor = random (16);
      putpixel(x, y, cor);
    } while (!kbhit());
 closegraph();
}
```

17.5 line

A função **line** (linhas) permite que uma linha seja plotada através de dois pontos Pi (xi, yi) e Pf (xf, yf).

Sintaxe: void **line** (int xi, int yi, int xf, int yf);
Prototype: graphics.h

Programa exemplo (59): O programa exibe linhas aleatórios na tela.

```
#include <graphics.h>
#include <conio.h>        /* kbhit */
#include <stdlib.h>       /* random */

void main (void)
{
 int placa, modo;
 int xi, yi, xf, yf, cor;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 randomize();
 do {
      xi = random (640);
      yi = random (480);
      xf = random (640);
      yf = random (480);
      cor = random (16);
      setcolor (cor);
      line(xi, yi, xf, yf);
     } while (!kbhit());
 closegraph();
}
```

17.6 rectangle

A função **rectangle** (retângulos) permite que um retângulo seja plotado na tela gráfica através de dois pontos Pi (xi,yi) e Pf (xf,yf). Estes dois pontos representam a diagonal do retângulo.

Sintaxe: void **rectangle** (int xi, int yi, int xf, int yf);
Prototype: graphics.h

Programa exemplo (60): O programa exibe retângulos aleatórios na tela.

```
#include <graphics.h>
#include <conio.h>        /* kbhit */
#include <stdlib.h>       /* random */

void main (void)
{
```

```
 int placa, modo;
 int xi, yi, xf, yf, cor;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 randomize();
 do {
      xi = random (640);
      yi = random (480);
      xf = random (640);
      yf = random (480);
      cor = random (16);
      setcolor (cor);
      rectangle (xi, yi, xf, yf);
    } while (!kbhit());
 closegraph();
}
```

17.7 circle

A função **circle** (círculos) permite que um círculo seja plotado na tela gráfica através do ponto central (Xc e Yc) e raio.

Sintaxe: void **circle** (int Xc, int Yc, int raio);
Prototype: graphics.h

Programa exemplo (61): O programa exibe círculos aleatórios na tela.

```
#include <graphics.h>
#include <conio.h>        /* kbhit */
#include <stdlib.h>       /* random */

void main (void)
{
 int placa, modo;
 int xc, yc, raio, cor;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 randomize();
 do {
      xc = random (640);
      yc = random (480);
      raio = random (50);
      cor = random (16);
      setcolor (cor);
      circle(xc, yc, raio);
    } while (!kbhit());
 closegraph();
}
```

17.8 arc

A função **arc** (arcos) permite desenhar um arco na tela gráfica através do ponto central (Xc, Yc), ângulo inicial, ângulo final e raio.

Sintaxe: void **arc** (int Xc, int Yc, int ang_inic, intang_fim, int raio);
Prototype: graphics.h

Programa exemplo (62): O programa exibe arcos aleatórios na tela.

```
#include <graphics.h>
#include <conio.h>        /* kbhit */
#include <stdlib.h>        /* random */

void main (void)
{
 int placa, modo;
 int xc, yc, raio, cor;
 int angulo_inicial, angulo_final;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 randomize();
 do {
      xc = random (640);
      yc = random (480);
      raio = random (20);
      angulo_inicial = random(360);
      angulo_final = random(360);
      cor = random (16);
      setcolor (cor);
      arc (xc, yc, angulo_inicial, angulo_final, raio);
     } while (!kbhit());
 closegraph();
}
```

17.9 drawpoly

A função **drawpoly** (polígonos) permite desenhar um polígono qualquer na tela.

Sintaxe: void **drawpoly** (int número_pontos, int vetor_pontos[]);
Prototype: graphics.h

Programa exemplo (63): O programa desenha uma *careta* na tela.

```
#include <graphics.h>
#include <conio.h>

void main (void)
```

```
{
 int rosto[18] = {109,149,209,149,259,124,259,74,209,39,109,39,59,74,59,124,109,149};
 int placa, modo;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
/* -------------------------------------- rosto */
 drawpoly (9, rosto);
/* -------------------------------------- olho esquerdo */
 circle(109, 74, 7);
/* -------------------------------------- olho direito */
 circle(209, 74, 7);
/* -------------------------------------- nariz */
 circle(159, 99, 15);
/* -------------------------------------- boca */
 rectangle(109, 120, 209, 128);
/* -------------------------------------- orelha esquerda */
 arc(59, 99, 90, 270, 20);
/* -------------------------------------- orelha direita */
 arc(259, 99, 270, 90, 20);
/* -------------------------------------- cabelos */
 arc(139, 39, 0, 105, 20);
 arc(179, 39, 75, 180, 20);
 while (!kbhit());
 closegraph();
}
```

Resultado do programa na tela:

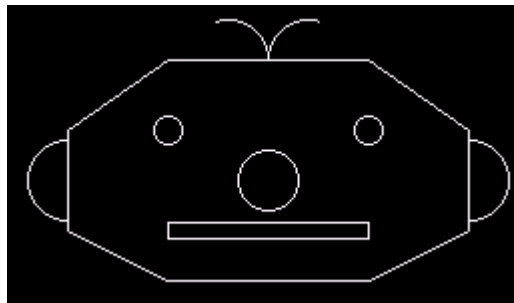



Figura 10: Rosto desenhado

17.10 setcolor e setbkcolor

setcolor: Função que permite a mudança da **cor da frente**, ou seja, a cor do objeto que será desenhado.

Sintaxe: void **setcolor** (int número_da_cor);
Prototype: graphics.h

setbkcolor: função que permite mudança da **cor de fundo** da tela.

Sintaxe: void **setbkcolor** (int número_da_cor);

Prototype: graphics.h

getcolor: Função que retorna a **cor da frente** (desenho) corrente.

Sintaxe: int **getcolor** (void);
Prototype: graphics.h

getbkcolor: Função que retorna a **cor de fundo** corrente.

Sintaxe: int **getbkcolor** (void);
Prototype: graphics.h

17.11 outtextxy e settextstyle

settextstyle: Modifica o **tipo de fonte** (caracter), **direção** e o **tamanho** do texto.

Sintaxe: void **settextstyle** (int fonte, int direção, int tamanho);
Prototype: graphics.h

Tabela 29: Tipos de fontes

Número	Fontes	Direções	Tamanhos
0	DEFAULT_FONT	HORIZ_DIR ou VERT_DIR	0..10
1	TRIPLEX_FONT	HORIZ_DIR ou VERT_DIR	0..10
2	SMALL_FONT	HORIZ_DIR ou VERT_DIR	0..10
3	SANS_SERIF_FONT	HORIZ_DIR ou VERT_DIR	0..10
4	GOTHIC_FONT	HORIZ_DIR ou VERT_DIR	0..10

A função **outtextxy** permite que um texto seja escrito na tela gráfica na posição (x, y).

Sintaxe: void **outtextxy** (int x, int y, char *texto);
Prototype: graphics.h

Programa exemplo (64): O programa exibe textos com fontes diferentes na tela.

```
#include <graphics.h>
#include <conio.h>

void main (void)
{
 int placa, modo;

 placa = VGA;
 modo = VGAHI;
 initgraph (&placa, &modo, "");
 setbkcolor (YELLOW);
 bar (0, 0, 319, 199);     /* bar desenha um retângulo cheio (pintado) */
 setcolor (GREEN);
 settextstyle (TRIPLEX_FONT, HORIZ_DIR, 5);   /* seta a fonte, direção e tamanho do texto*/
```

```
 outtextxy (10, 10, "Turbo C V2.01");           /* imprime texto na tela gráfica */
 setcolor (RED);
 settextstyle (SANS_SERIF_FONT, HORIZ_DIR, 4);
 outtextxy (10, 50, "Turbo C V2.01");
 setcolor (BLACK);
 settextstyle (SMALL_FONT, HORIZ_DIR, 7);
 outtextxy (10, 90, "Turbo C V2.01");
 while (!kbhit());                 /* pausa até que o usuário digite uma tecla qualquer */
}
```

Atenção: Para imprimir números inteiros ou reais (**int** ou **float**), deve-se convertê-los para **string**. Verifique o exemplo abaixo.

Exemplo: O programa imprime um número inteiro (**int**), um real positivo (**float**) e um real negativo (**float**).

```
#include <graphics.h>
#include <stdlib.h>

void insere_virgula_na_string (char *s, int casas, int sinal);

void main (void)
{
 int placa = DETECT, modo;
 int inteiro = 123;
 float float_pos = 123.45, float_neg = -1234.56;
 char s[10];
 int casas, sinal;

 initgraph(&placa, &modo, "d:\lingua~1\tc\fontes");

/* exibição de um inteiro */

 itoa(inteiro, s, 10);    /* 10 → decimal */
 settextstyle(SMALL_FONT, HORIZ_DIR,10);
 outtextxy(10, 10, s);

/* exibição de um float positivo */

 strcpy(s,ecvt(float_pos, 5, &casas, &sinal));      /* ecvt converte de float para string */
 insere_virgula_na_string(s, casas, sinal);
 outtextxy(10, 50, s);

 /* exibição de um float negativo */

 strcpy(s,ecvt(float_neg, 6, &casas, &sinal));
 insere_virgula_na_string(s, casas, sinal);
 outtextxy(10, 80, s);

 getch();
}

/* ---------------------------------- insere_virgula_na_string */

void insere_virgula_na_string (char *s, int casas, int sinal)
```

```
{
 int i, n, t;

 n = strlen(s);
 for (i = n; i > casas; i--)
     s[i] = s[i-1];
 s[casas] = ',';
 s[n+1] = NULL;
 if (sinal == 1)                /* float negativo */
    {
     n = strlen(s);
     for (i = n; i > 0; i--)
            s[i] = s[i-1];
     s[0] = '-';
     s[n+1] = NULL;
    }
}
```

17.12 Preenchimento

17.12.1 Pintura de retângulos

Sintaxe: void **bar** (int xi, int yi, int xf, int yf);
Prototype: graphics.h

Para preencher (pintar) um retângulo deve-se definir o estilo de preenchimento, utilizando a função:

Sintaxe: void **setfillstyle** (int estilo, int cor);
Prototype: graphics.h

Tabela 30: Estilos de preenchimento

Número	Nome do estilo
0	EMPTY_FILL
1	SOLID_FILL
2	LINE_FILL
3	LTSLASH_FILL
4	SLASH_FILL
5	BKSLASH_FILL
6	LTBKSLASH_FILL
7	HATCH_FILL
8	XHATCH_FILL
9	INTERLEAVE_FILL
10	WIDE_DOT_FILL
11	CLOSE_DOT_FILL
12	USER_FILL

EMPTY_FILL: Preenche com a cor de fundo.

Programa exemplo (65): O programa desenha doze retângulos cheios mostrando os estilos de preenchimento.

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

void main (void)
{
 int placa, modo;
 unsigned int estilo;
 int cor = 1;

 placa = VGA;
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 for (estilo = 1; estilo <= 12; estilo++)
     {
      setfillstyle (estilo, cor);
      bar (0, 0, 100, 100);
      cor++;
      if (cor > 15)
         cor = 1;
      getch();
    }
 closegraph();
}
```

17.12.2 Pintura de polígonos

a) **fillpoly**: Desenha um polígono preenchido com um estilo selecionado pela função **setfillstyle**.

Sintaxe: void **fillpoly** (int estilo, int cor);
Prototype: graphics.h

Para preencher um polígono a função **fillpoly** utiliza o estilo definido pela função **setfillstyle** descrito anteriormente. Esta função preenche qualquer polígono fechado, se o polígono estiver aberto esta função preenche até encontrar um objeto fechado (**isto é normalmente um erro grave**).

Programa exemplo (66): O programa exibe um objeto preenchido com doze estilos diferentes.

```
#include <graphics.h>
#include <conio.h>

void main (void)
{
 int objeto[10] = {159, 0, 0, 50, 159, 199, 319, 50, 159,0};
 int placa, modo, estilo, cor = 1;

 placa = VGA;
```

```
 modo = VGAHI;
 initgraph(&placa, &modo, "");
 setbkcolor(WHITE);
 setcolor(2);                                    /* seta cor do desenho */
 for (estilo = 0; estilo <= 12; estilo++)
    {
     setfillstyle (estilo, cor);                 /* define estilo e cor */
     fillpoly (5, objeto);                       /* pinta polígono */
     cor++;
     if (cor > 15)
       cor = 1;
     getch();
    }
 closegraph();
}
```

Resultado do programa na tela:

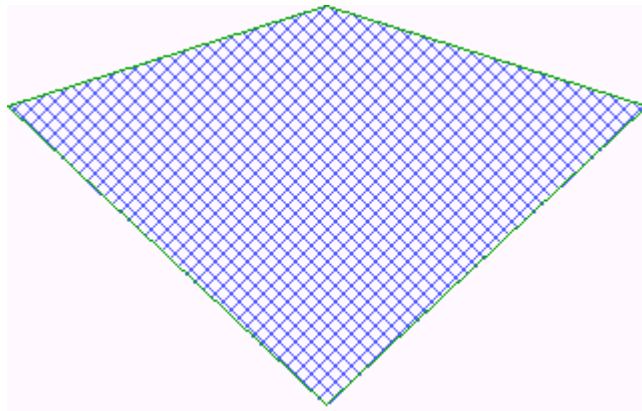



Figura 11: Polígono preenchido

b) **floodfill**: Função preenche um polígono ao redor de um ponto (função recursiva).

Sintaxe: void **floodfill** (int x, int y, int cor_da_borda);
Prototype: graphics.h

Para preencher um polígono a função **floodfill** utiliza o estilo definido pela função **setfillstyle** descrito anteriormente. A função **floodfill** parte de um ponto central (x, y) do objeto e preenche ao redor deste ponto até encontrar uma borda com a cor especificada.

Programa exemplo (67): O programa exibe uma espécie de rosa dos ventos.

```
#include <graphics.h>
#include <conio.h>

void main (void)
{
 int rosa[34] = {159, 1, 189, 49, 279, 24, 239, 72, 319, 99, 239, 123, 279, 173, 189,
                 149, 159, 198, 109, 149, 39, 173, 79, 123, 1, 99, 79, 72, 39, 24, 109, 49, 159, 1};
```

```
 int placa, modo, estilo, cor = 1;

 placa = VGA;
 modo = VGAHI;
 initgraph (&placa, &modo, "");
 setbkcolor (WHITE);
 for (estilo = 0;estilo <= 12;estilo++)
    {
     setfillstyle (0, 0);
     bar (0, 0, 319, 199);
     setcolor (GREEN);                    /* cor da borda */
     drawpoly (17, rosa);                 /* imprime rosa_dos_ventos */
     circle (159, 99, 20);                /* imprime círculo */
     setfillstyle (estilo, cor);          /* define estilo e cor */
     floodfill (10, 99, GREEN);           /* preenchimento */
     cor++;
     if (cor > 15)
        cor = 1;
     getch();
    }
closegraph ();
}
```

Resultado do programa na tela:

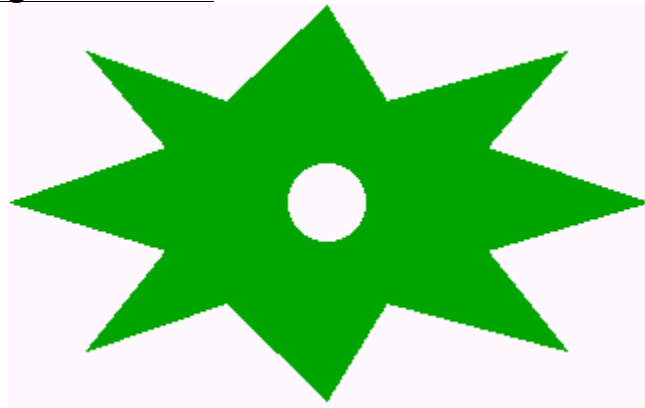



Figura 12: Polígono preenchido

17.13 Ativação de janelas gráficas

17.13.1 Janela ativa

Permite definir uma área da tela como janela ativa, ou seja, nada pode ser desenhado fora desta área.

Sintaxe: void **setviewport** (int xi, int yi, int xf, int yf, int recorta);
Prototype: graphics.h

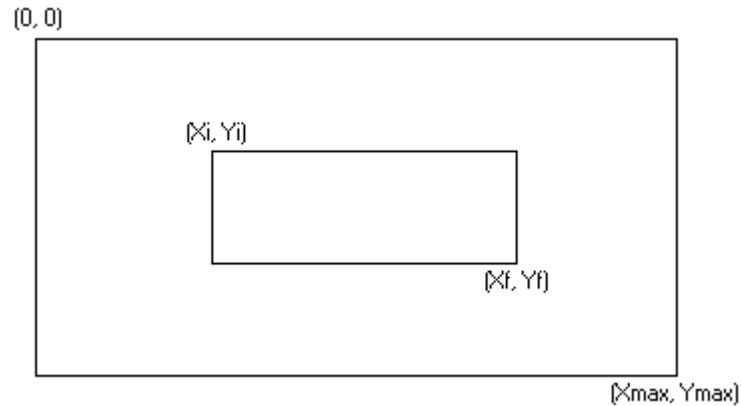



Figura 13: Coordenadas de uma janela

(xi, yi): ponto inicial
(xf, yf): ponto final
recorta: **true** (!0) ou **false** (0) - variável indica se haverá recorte na janela ou não.

Programa exemplo (68): O programa define uma janela ativa colocando um minicursor que se movimenta dentro desta janela.

```
#include <graphics.h>
#include <conio.h>

#define ENTER       13
#define ESC         27

#define UP          72
#define DOWN        80
#define LEFT        75
#define RIGHT       77

#define TRUE        !0
#define FALSE       0

void main (void)
{
 int placa, modo;
 int x, y;
 char tecla;

 placa = VGA;
 modo = VGAHI;
 initgraph (&placa, &modo, "");
 setfillstyle (1,1);
 bar (0, 0, 319, 199);
 setviewport (19,19,300,180,TRUE);          /* teste usando FALSE */
 setcolor (2);
 clearviewport ();
 x = 159;
 y = 99;
 do {
    setcolor (2);
```

```
        rectangle (x, y, x+10, y+10);
        tecla = getch();
        setcolor (0);
        rectangle (x, y, x+10, y+10);
        switch (tecla)
        {
              case UP: y = y - 5;
                       break;
          case DOWN: y = y + 5;
                       break;
            case LEFT:  x = x - 5;
                       break;
          case RIGHT:  x = x + 5;
                       break;
        }
      } while (tecla != ENTER && tecla != ESC);
}
```

<u>Resultado do programa na tela</u>:

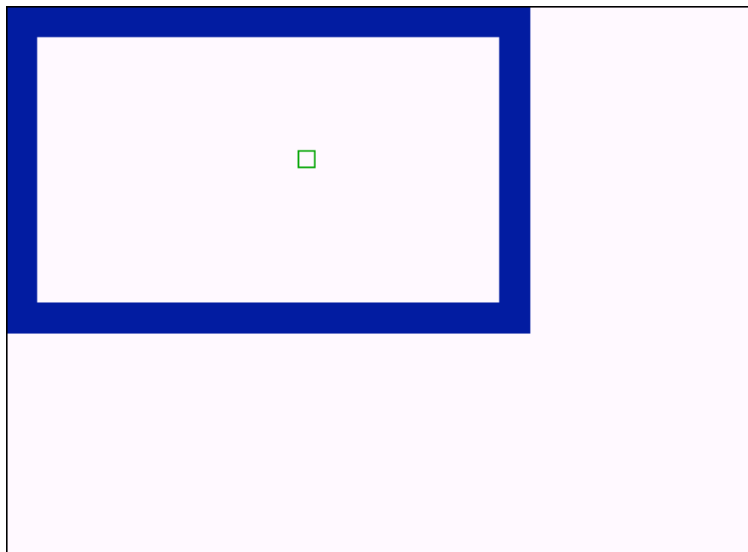



Figura 14: Janela ativa

17.13.2 Limpar janela ativa

Sintaxe: void **clearviewport** (void);
Prototype: graphics.h

Função que limpa a janela ativada pela função **setviewport**, se nenhuma janela for ativada, por falta (**default**), a janela ativa é toda a tela.

17.13.3 Salvar e recuperar janelas gráficas

Quando é necessário colocar janelas sobrepostas (ou outras informações) na tela, deve-se, antes, salvar a área da colocação da nova janela.

Para salvar e recuperar janelas na memória RAM (áreas da tela) são utilizados os seguintes procedimentos:

A função **getimage** armazena na memória uma área da tela.

Sintaxe: int **getimage** (int xi, int yi, int xf, int yf, void *pointer);
Prototype: graphics.h

A função **putimage** carrega para a tela uma área da memória que contém uma porção de memória armazenada anteriormente pela função **getimage**.

Sintaxe: int **putimage** (int xi, int yi, void *pointer, int modo);
Prototype: graphics.h

Tabela 31: Modos de exibição

Modos	Efeito
COPY_PUT	**copia** imagem
XOR_PUT	**ou exclusivo** com a imagem contida na tela
OR_PUT	**ou** com a imagem contida na tela
AND_PUT	**e** com a imagem contida na tela
NOT_PUT	**não** com a imagem contida na tela

Para definir o ponteiro que irá conter os **pixels** da tela, é necessário saber a quantidade de bytes (memória) para salvar a janela gráfica, isto é feito utilizando-se a função **imagesize**.

Sintaxe: long int **imagesize** (int xi, int yi, int xf, int yf);
Prototype: graphics.h

A função **imagesize** retorna o número de bytes necessários para armazenar a janela definida pelos pontos **Pi** (xi, yi) e **Pf** (xf, yf).

Sintaxe: void ***malloc**(int número_bytes);
Prototype: alloc.h e stdlib.h

A função **malloc** (**m**emory **alloc**ation) permite alocação dinânica de memória para o ponteiro **pointer**, que ocupará **n** bytes na memória RAM.

Sintaxe: void **free** (void *pointer);
Prototype: alloc.h e stdlib.h

A função **free** libera uma área de memória alocada para o ponteiro **pointer**.

Programa exemplo (69): O programa exibe três janelas sobrepostas na tela, permitindo retira-las sem apagar o conteúdo das informações que estão por baixo.

```
#include <graphics.h>
```

```
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>

#define JANELAS 10

void moldura (int xi, int yi, int xf, int yf, int frente, int fundo);
void salva_janela (int janela, int xi, int yi, int xf, int yf);
void restaura_janela (int janela, int xi, int yi, int xf, int yf);

unsigned int numero_de_bytes[JANELAS];
void *p[JANELAS];

void main (void)
{
 int placa, modo;
 int cor;

 placa = VGA;
 modo = VGAHI;
 initgraph (&placa, &modo, "");
 setfillstyle (0, 0);
 bar (0, 0, 639, 479);
 salva_janela (1, 10, 10, 100, 50);
 moldura (10, 10, 100, 50, BLUE, LIGHTBLUE);
 getch();
 salva_janela (2, 30, 30, 130, 80);
 moldura (30, 30, 130, 80, GREEN, LIGHTGREEN);
 getch();
 salva_janela (3, 50, 50, 150, 100);
 moldura (50, 50, 150, 100, RED, REDLIGHT);
 getch();
 restaura_janela (3, 50, 50, 150, 100);
 getch();
 restaura_janela (2, 30, 30, 130, 80);
 getch();
 restaura_janela (1, 10, 10, 100, 50);
 getch();
 closegraph();
}

void moldura (int xi, int yi, int xf, int yf, int frente, int fundo)
{
 setcolor (fundo);
 setfillstyle (SOLID_FILL, fundo);
 bar (xi, yi, xf, yf);
 setcolor (frente);
 rectangle (xi, yi, xf, yf);
 rectangle (xi+2, yi+2, xf-2, yf-2);
}

void salva_janela (int janela, int xi, int yi, int xf, int yf)
{
 numero_de_bytes[janela] = imagesize (xi, yi, xf, yf);
 p[janela] = (void *) malloc (numero_de_bytes[janela]);
 getimage (xi, yi, xf, yf, p[janela]);
```

```
}

void restaura_janela (int janela, int xi, int yi, int xf, int yf)
{
 setfillstyle (0, 0);
 bar (xi, yi, xf, yf);
 putimage (xi, yi, p[janela], XOR_PUT);
 free (p[janela]);
}
```

Resultado do programa na tela:

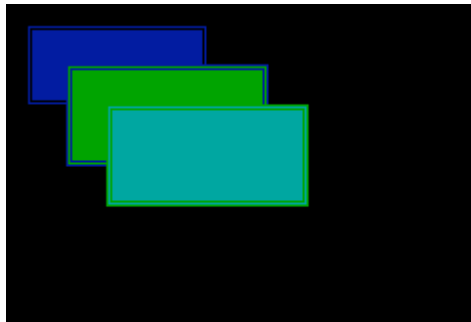



Figura 15: Sobreposição de janelas

17.14 - Lista de exercícios (gráficos, arquivos, estruturas e campos de bits)

- ✓ Escreva um programa em C (**grava.c**) que desenha círculos coloridos aleatórios na tela em modo gráfico, com raio igual à 5. Quando o usuário digitar alguma tecla, grave toda a imagem gerada em um arquivo chamado "**imagem.pic**".

Observação: Note que o arquivo gerado (**imagem.pic**) possui 307.200 bytes, pois a cor de cada pixel é armazenada no arquivo com 1 byte (tipo **char**) ou seja, 16 bits. A resolução da tela é 640 x 480 (placa = **VGA**, modo = **VGAHI**) totalizando 307.200 *pixels*.

- ✓ Escreva um programa em C (**exibe.c**) que lê o arquivo "**imagem.pic**" e exibe na tela a imagem gravada neste arquivo, ou seja, a imagem que foi gerada e armazenada pelo programa anterior.

- ✓ Re-escreva o programa "**bits.c**" de forma que o novo arquivo gerado (**imagem.bit**) ocupe apenas a metade do espaço utilizado pelo arquivo "**imagem.pic**".

Observação: Isto pode ser conseguido utilizando uma estrutura com dois campos de bits, ou seja, cada um com 4 bits. Note que utilizando a placa VGA e modo VGAHI, temos apenas 16 cores. Para armazenar 16 cores é necessário apenas 4 bits e não 8 como foi utilizado anteriormente. Esta economia de bits fará com que o arquivo ocupe apenas 153.600 bytes.

✓ Agora re-escreva o programa "**exibe.c**" de forma a mostrar na tela o arquivo "**imagem.bit**" gerado pelo programa "**bits.c**". Este programa chama-se "**mostra.c**".

18. Memória de vídeo

Em C pode-se escrever diretamente na memória de vídeo sem utilizar o comando **printf**, isto é feito endereçando um vetor de 4000 bytes a partir do endereço **0xb8000000** (endereço na placa de vídeo). Este endereço é o início da tela em modo texto. Isto é feito através de um ponteiro que aponta diretamente para o endereço da memória de vídeo.

Como foi visto anteriormente, um ponteiro em C pode ser indexado, ou seja, a primeira posição da tela está em **p[0]**, a segunda em **p[2]**, pois **p[1]** é o atributo de cor do primeiro caracter.

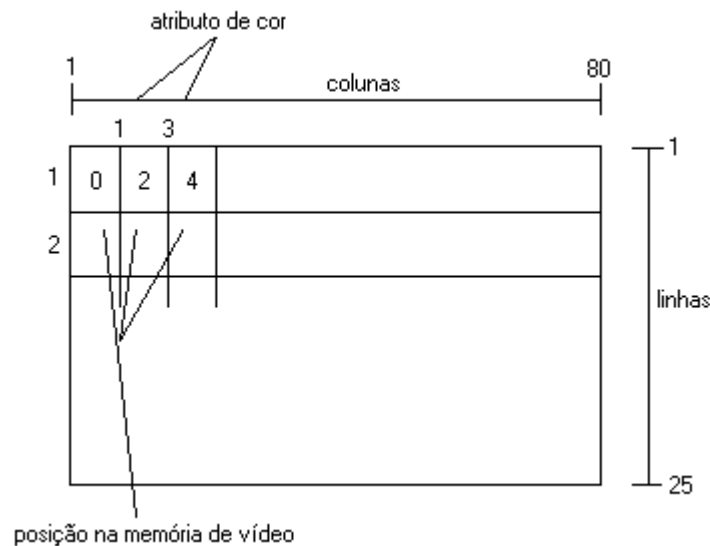



Figura 16: Memória de vídeo (atributos de tela)

Localização na memória de vídeo

A tela é bidimensional, ou seja, para imprimir um caracter na tela é necessário utilizar duas variáveis: **coluna** e **linha** (*gotoxy (col, lin);*). O ponteiro utilizado para apontar para a memória de vídeo é unidimensional, ou seja, precisa apenas de uma variável: **índice do vetor**. O cálculo deste índice é feito através da posição: **coluna** e **linha**. O cálculo é:

índice = 160 * (linha - 1) + 2 * (coluna - 2);

Atributo COR: (1 byte)

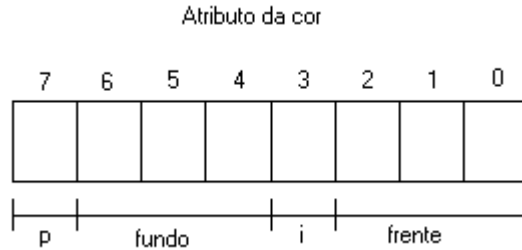



Figura 17: Atributos da cor

Onde: **p**: caracter piscante
i: intensidade da cor
fundo: cor de fundo (3 bits, total 8 cores)
frente: cor de frente (4 bits, total 16 cores)

Programa exemplo (70): O programa escreve diretamente na memória de vídeo.

```
#include <stdio.h>
#include <conio.h>

#define UP          72
#define DOWN        80
#define LEFT        75
#define RIGHT       77
#define HOME        71
#define END         79
#define PGUP        73
#define PGDN        81

#define ENTER       13          /* seleciona o início e fim da moldura */
#define ESC         27

#define F1          59          /* moldura SIMPLES */
#define F2          60          /* moldura DUPLA */

#define TRUE        !0
#define FALSE       0

#define CURSOR      219

void retangulo (char c, int xi, int yi, int xf, int yf);
void imprime (int c, int l, char letra);
char recupera (int c, int l);
void inverte (int *xi, int *yi, int *xf, int *yf);

char far *p = (char far *) 0xb8000000l;          /* l pois o número é long */

void main (void)
{
 int c = 20, l = 12, ct = 20, lt = 12;
 int xi, yi, xf, yf;
 char tecla, car = 'S';
 int primeiro = TRUE;
 char caracter = 32;
```

```
textbackground(BLACK);
window(1, 1, 80, 25);
clrscr();
gotoxy(1, 1);
printf(" Coluna: %2d", c);
gotoxy(1, 2);
printf("  Linha: %2d", l);
gotoxy(1, 3);
if (car == 'S')
      printf("Moldura: SIMPLES");
else
      printf("Moldura: DUPLA  ");
do {
    gotoxy(10, 1);
    printf("%2d", c);
    gotoxy(10, 2);
    printf("%2d", l);
    gotoxy(10, 3);
    if (car == 'S')
          printf("SIMPLES");
    else
          printf("DUPLA  ");
    ct = c;
    lt = l;
    caracter = recupera (ct, lt);
    imprime(c, l, CURSOR);
    tecla = getch();
    if (tecla == 0 || tecla == ENTER)
       {
          if (tecla != ENTER)
             tecla = getch();
          imprime(c, l, 32);
          imprime(c, l, caracter);
          switch (tecla)
                    {
                       case UP: l--;
                                      if (l < 1)
                                         l = 25;
                                       break;
                 case DOWN: l++;
                                      if (l > 25)
                                         l = 1;
                                      break;
                   case LEFT: c--;
                                      if (c < 1)
                                         c = 80;
                                      break;
                 case RIGHT: c++;
                                      if (c > 80)
                                         c = 1;
                                      break;
                 case HOME: c = 1;   /* canto superior esquerdo */
                                      l = 1;
                                      break;
                    case END: c = 1;   /* canto inferior esquerdo */
```

```
                l = 25;
                break;
        case PGUP: c = 80;/* canto superior direito */
                l = 1;
                break;
        case PGDN: c = 80;/* canto inferior direito */
                l = 25;
                break;
          case F1: car = 'S';     /* seleciona moldura simples */
                break;
          case F2: car = 'D';     /* seleciona moldura dupla */
                break;
       case ENTER: if (primeiro)
                 {
                  xi = c;           /* marca o canto superior esquerdo da moldura */
                  yi = l;
                  primeiro = FALSE;
                 }
                else
                   {
                    xf = c;   /* desenha a moldura */
                    yf = l;
                    inverte(&xi, &yi, &xf, &yf);        /* inverte os cantos */
                    retangulo(car, xi, yi, xf, yf);
                    primeiro = TRUE;
                    c = (xi + xf) / 2;
                    l = (yi + yf) / 2;
                   }
                break;
        }
   }
  } while (tecla != ESC);
}

/* ---------------------------------------------------------- retangulo */

void retangulo(char c, int xi, int yi, int xf, int yf)
{
 int col, lin;

 if (c == 'D')      /* moldura DUPLA */
   {
    for (col = xi; col <= xf; col++)
       {
        imprime(col, yi, 205);
        imprime(col, yf, 205);
       }
    for (lin = yi; lin <= yf; lin++)
       {
        imprime(xi, lin, 186);
        imprime(xf, lin, 186);
       }
    imprime(xi, yi, 201);
    imprime(xf, yi, 187);
    imprime(xf, yf, 188);
    imprime(xi, yf, 200);
```

```
    }
  else
      {
       for (col = xi; col <= xf; col++)
           {
             imprime(col, yi, 196);
             imprime(col, yf, 196);
           }
      for (lin = yi; lin <= yf; lin++)
         {
           imprime(xi, lin, 179);
           imprime(xf, lin, 179);
         }
      imprime(xi, yi, 218);
      imprime(xf, yi, 191);
      imprime(xf, yf, 217);
      imprime(xi, yf, 192);
     }
}

/* ------------------------------------------------------- imprime */

void imprime (int c, int l, char letra)
{
 long int indice;

 indice = 160 * (l - 1) + 2 * (c - 1);
 p[indice] = letra;                                /* imprime diretamente na memória de vídeo */
 gotoxy(c, l);
}

/* ------------------------------------------------------- recupera */

char recupera (int c, int l)
{
 long int indice;
 char car;

 indice = 160 * (l-1) + 2 * (c-1);
 car = (char) p[indice];              /* recupera um caracter diretamente da memória de vídeo */
 return(car);
}

/* --------------------------------------------------------------- inverte */

void inverte (int *xi, int *yi, int *xf, int *yf)
{
 int temp;

 if (*xi > *xf)
    {
     temp = *xi;
     *xi = *xf;
     *xf = temp;
    }
 if (*yi > *yf)
```

```
  {
   temp = *yi;
   *yi = *yf;
   *yf = temp;
  }
}
```

Resultado do programa na tela:

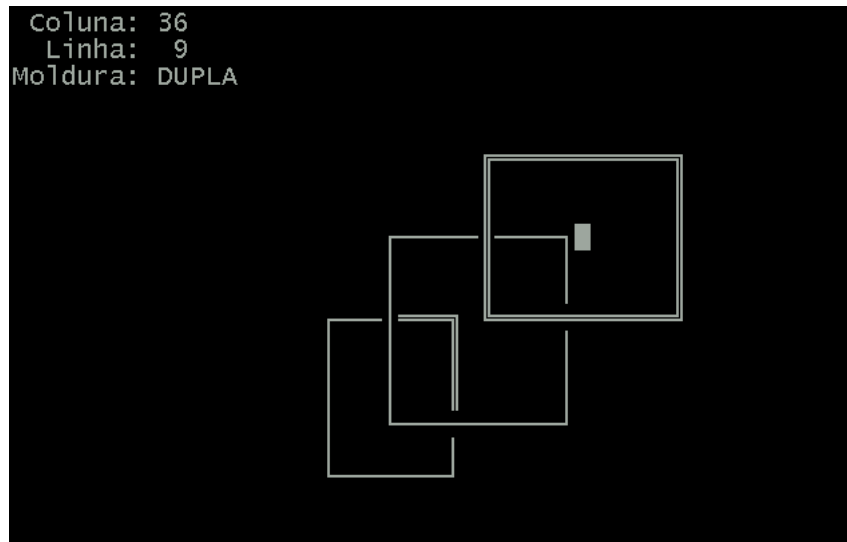



Figura 18: Utilização da memória de vídeo via ponteiro

19. Interrupções

Um computador PC (**Personal Computer**) possui um circuito integrado responsável pela interrupção do processamento do microprocessador principal (8086) chamado **PIC 8259** (**P**rogrammable **I**nterrupt **C**ontroller - controlador de Interrupções programado), possuindo **256** tipos de interrupções diferentes.

As interrupções podem ser de **hardware** ou **software**. Por exemplo **Ctrl + Break** é uma interrupção de **hardware** via teclado.

Função de interrupção

Sintaxe: int **int86** (int número_da_interrupção, union REGS *regs_in, union REGS *regs_out);

Observação: **regs_in** e **regs_out** podem ser a mesma variável.

union REGS: É um tipo de dado pré-definido do C que permite manipular diretamente os registradores do microprocessador.

Atenção: Para utilizar esta função é necessário inserir: **#include <dos.h>**

Programa exemplo (71): O programa possui uma função que permite posicionar o cursor na tela, ou seja, o programa possui uma função: **GOTOXY (c, l)**;

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

void GOTOXY (int c, int l);

void main (void)
{
 clrscr();
 GOTOXY(38,1);
 printf("UCPel");
 getch();
}

/* ------------------------------------ GOTOXY  */

void GOTOXY (int c, int l)
{
 union REGS regs;

 regs.h.ah = 2;
 regs.h.dl = c - 1;
 regs.h.dh = l - 1;
 regs.h.bh = 0;
 int86(0x10, &regs, &regs);
}
```

Programa exemplo (72): O programa testa se uma impressora (**IBM 2390**) está **on-line** (ligada) ou **off-line** (desligada)

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

int impressora_ligada (void);

void main (void)
{
 clrscr();
 do {
      if (impressora_ligada())
            printf("Atenção: Impressora está ligada\n");
      else
            printf("Atenção: Impressora está desligada\n");
   } while (!kbhit());
}

/* ----------------------------------- impressora_ligada */

int impressora_ligada (void)
{
 union REGS regs;
```

```
 regs.h.ah = 0x02;
 regs.x.dx = 0x00;
 int86(0x17, &regs, &regs);
 if (regs.h.ah == 144)
        return (1);
 else
       return (0);
}
```

Programa exemplo (73): O programa utiliza o **mouse** em modo texto.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

void inicializa_mouse (void);
void mostra_cursor_mouse (void);
void apaga_cursor_mouse (void);
void le_posicao_cursor_mouse (int *xmouse, int *ymouse, int *botao);
void posiciona_cursor_mouse (int x, int y);

#define ESQUERDO     1
#define DIREITO          2

int xmouse, ymouse, botao;

void main(void)
{
 clrscr();
 inicializa_mouse();
 posiciona_cursor_mouse(40, 12);
 mostra_cursor_mouse();
 le_posicao_cursor_mouse(&xmouse, &ymouse, &botao);
 gotoxy(1, 1);
 printf("x: %02d", xmouse);
 gotoxy(1,2);
 printf("y: %02d", ymouse);
 gotoxy(1,3);
 printf("b: %d", botao);
 do {
       le_posicao_cursor_mouse(&xmouse, &ymouse, &botao);
       if (botao == ESQUERDO)
          {
           gotoxy(xmouse, ymouse);
           printf("Ok");
           delay(500);
           gotoxy(xmouse, ymouse);
           clreol();
          }
     gotoxy(1,1);
     printf("x: %02d", xmouse);
     gotoxy(1,2);
     printf("y: %02d", ymouse);
     gotoxy(1,3);
```

```
    printf("b: %d", botao);
   } while (botao != DIREITO);
}

/* ---------------------------- inicializa_mouse */

void inicializa_mouse (void)
{
 union REGS regs;

 regs.x.ax = 0x0;
 int86(0x0033, &regs, &regs);
}

/* ---------------------------- mostra_cursor_mouse */

void mostra_cursor_mouse (void)
{
 union REGS regs;

 regs.x.ax = 0x1;
 int86(0x0033, &regs, &regs);
}

/* ---------------------------- apaga_cursor_mouse */

void apaga_cursor_mouse (void)
{
 union REGS regs;

 regs.x.ax = 0x2;
 int86(0x0033, &regs, &regs);
}

/* --------------------------- le_posicao_cursor_mouse */

void le_posicao_cursor_mouse (int *xmouse, int *ymouse, int *botao)
{
 union REGS regs;

 regs.x.ax = 0x3;
 int86(0x0033, &regs, &regs);
 *botao = regs.x.bx;
 *xmouse = 1 + (regs.x.cx / 8);
 *ymouse = 1 + (regs.x.dx / 8);
}

/* --------------------------- posiciona_cursor_mouse */

void posiciona_cursor_mouse (int x, int y)
{
 union REGS regs;

 regs.x.cx = (x * 8) - 1;
 regs.x.dx = (y * 8) - 1;
 regs.x.ax = 0x4;
```

```
 int86(0x0033, &regs, &regs);
}
```

Programa exemplo (74): O programa utiliza o **mouse** em modo gráfico.

```
#include <stdio.h>
#include <dos.h>
#include <graphics.h>

void inicializa_tela_grafica (void);
void finaliza_tela_grafica (void);
void inicializa_mouse (void);
void mostra_cursor_mouse (void);
void apaga_cursor_mouse (void);
void le_posicao_cursor_mouse (int *xmouse, int *ymouse, int *botao);
void posiciona_cursor_mouse (int x, int y);

#define ESQUERDO     1
#define DIREITO      2

int xmouse, ymouse, botao;

void main (void)
{
 int raio = 50;

 inicializa_tela_grafica();
 inicializa_mouse();
 posiciona_cursor_mouse(40,12);
 mostra_cursor_mouse();
 do {
       le_posicao_cursor_mouse(&xmouse, &ymouse, &botao);
       if (botao == ESQUERDO)
          {
           setcolor(WHITE);
           circle(xmouse, ymouse, raio);
           delay(500);
           setcolor(BLACK);
           circle(xmouse, ymouse, raio);
          }
    } while (botao != DIREITO);
 finaliza_tela_grafica();
}

/* ----------------------------- inicializa_tela_gráfica */

void inicializa_tela_gráfica (void)
{
 int placa, modo;

 placa = DETECT;
 initgraph(&placa, &modo, "d:\\tc\\bgi");
}

/* ----------------------------- finaliza_tela_gráfica */
```

```
void finaliza_tela_gráfica (void)
{
 closegraph();
}

/* ---------------------------- inicializa_mouse */

void inicializa_mouse (void)
{
 union REGS regs;

 regs.x.ax = 0x0;
 int86(0x0033, &regs, &regs);
}

/* ---------------------------- mostra_cursor_mouse */

void mostra_cursor_mouse (void)
{
 union REGS regs;

 regs.x.ax = 0x1;
 int86(0x0033, &regs, &regs);
}

/* ---------------------------- apaga_cursor_mouse */

void apaga_cursor_mouse (void)
{
 union REGS regs;

 regs.x.ax = 0x2;
 int86(0x0033, &regs, &regs);
}

/* --------------------------- le_posicao_cursor_mouse */

void le_posicao_cursor_mouse (int *xmouse, int *ymouse, int *botao)
{
 union REGS regs;

 regs.x.ax = 0x3;
 int86(0x0033, &regs, &regs);
 *botao = regs.x.bx;
 *xmouse = regs.x.cx;
 *ymouse = regs.x.dx;
}

/* -------------------------- posiciona_cursor_mouse */

void posiciona_cursor_mouse (int x, int y)
{
 union REGS regs;

 regs.x.cx = x;
 regs.x.dx = y;
```

```
 regs.x.ax = 0x4;
 int86(0x0033, &regs, &regs);
}
```

20. Listas lineares

As listas lineares encadeadas (**filas** e **pilhas** alocadas dinamicamente) permitem alocação indeterminada de elementos em uma estrutura de dados. Os elementos são alocados na memória RAM dinamicamente.

Uma lista encadeada tem por característica um **elo** de ligação entre um elemento e outro. Ao contrário de um vetor, onde os elementos são alocados estaticamente e em posições contíguas na memória RAM, uma lista encadeada é alocada dinamicamente, tendo como característica básica, que os elementos são alocados em posições diferentes (aleatórias) na memória.

As listas lineares encadeadas possuem um **header** (cabeça) que armazena o primeiro elemento da lista.

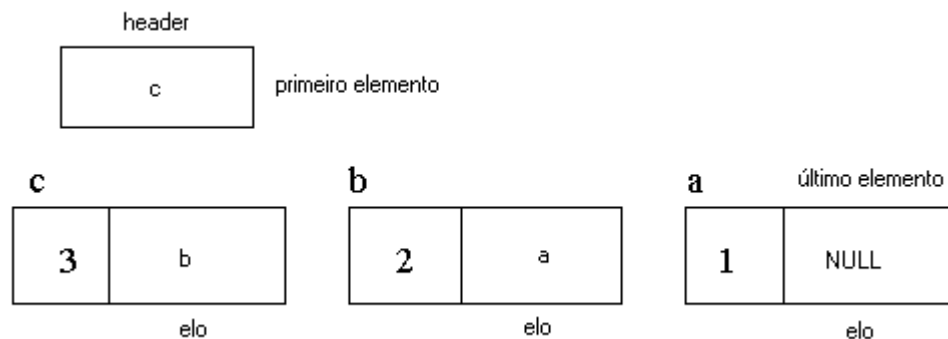
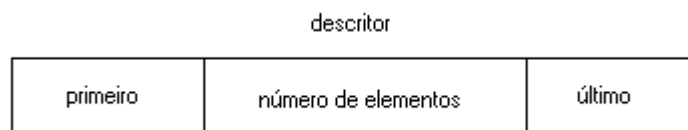



Figura 19: Representação de uma lista encadeada

Dois problemas existem em uma lista encadeada:

a) **Primeiro**: É que as listas são percorridas seqüencialmente, apenas numa direção, ou seja, do primeiro ao último elemento;
b) **Segundo**: É que a informação do número de elementos da lista é obtido somente por uma varredura completa na lista.

Para resolver estes dois problemas pode-se utilizar um descritor da seguinte maneira:

As vantagens de um descritor são:

- Conhecer o número de elementos da lista linear sem ter que percorrê-la;
- Acessar o último elemento facilitando a inserção ou remoção no final da lista.

As listas podem ser organizadas como: **pilhas** ou **filas.**

Pilha: Estrutura linear organizada de forma que a entrada e a saída dos dados é feita na mesma extremidade.

Forma de acesso: LIFO (**L**ast **I**nput **F**irst **O**utput), ou seja, o último elemento a entrar na pilha é o primeiro a sair dela.

Fila: Estrutura linear organizada em forma que a entrada dos dados é feita por uma extremidade da lista linear e, a saída, é feita na outra extremidade.

Forma de acesso: FIFO (**F**irst **I**nput **F**irst **O**utput), ou seja, o primeiro elemento a entrar na fila é o primeiro a sair da fila.

E - Entrada de Dados
S - Saída de Dados

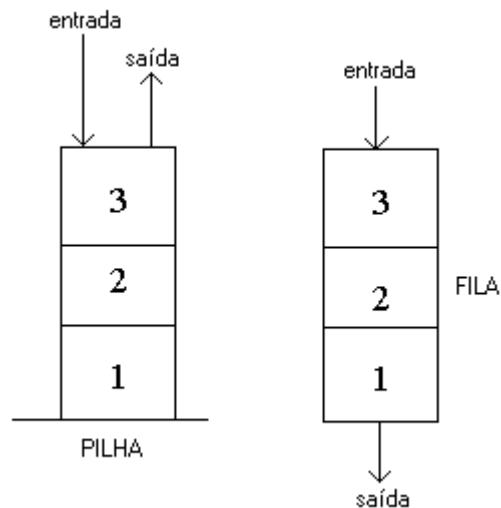



Figura 20: Representação de uma fila e uma pilha

Funcionamento desta pilha:

Entrada: 1, 2 e 3
Saída: 3, 2 e 1

Funcionamento desta fila:

Entrada: 1, 2 e 3
Saída: 1, 2 e 3

20.1 - Implementação de uma pilha

Programa exemplo (75): O programa permite inserir números inteiros em uma pilha. Quando o número digitado for igual à zero (0), todos os números da pilha são listados.

```
// pilha.c

#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <stdlib.h>
#include <string.h>

#define SUCESSO                         1
#define FALTA_DE_MEMORIA                2
#define PILHA_VAZIA                     3

struct DADO {
                int info;                               /* informação */
                struct DADO elo;                / elo para o próximo elemento */
               };

struct PILHA {
                struct DADO *topo;
               };

void cria_pilha (struct PILHA *p);
int push (struct PILHA *p, int valor);
int pop (struct PILHA *p, int *info);
int consulta_pilha (struct PILHA p, int *info);
void imprime_erro (int erro);

/* ----------------------------------- main */

void main (void)
{
 struct PILHA p;
 int valor, erro;
 char ch;

 cria_pilha(&p);
 do {
      clrscr();
      printf("[P]ush, P[O]p, [C]onsultar ou [F]im ?");
      do {
            ch = toupper(getche());
          } while (!strchr("POCF", ch));
     if (ch == 'P')
       {
         printf("Valor: ");
         scanf("%d", &valor);
       }
     switch (ch)
        {
          case 'P': erro = push(&p, valor);
                       if (erro != SUCESSO)
                          imprime_erro(erro);
                       break;
```

```
        case 'O': erro = pop(&p,&valor);
                    if (erro != SUCESSO)
                        imprime_erro(erro);
                      else
                         printf("Informação Excluída: %d\n",valor);
                    break;
        case 'C': erro = consulta_pilha(p, &valor);
                    if (erro != SUCESSO)
                       imprime_erro(erro);
                    else
                         printf("Valor: %d\n", valor);
                   break;
     }
    } while (ch != 'F');
}

/* ------------------------------------ cria_pilha */

void cria_pilha (struct PILHA *p)
{
 p->topo = NULL;
}

/* ------------------------------------ push */

int push (struct PILHA *p, int valor)
{
 struct DADO *t;

 t = (struct DADO *) malloc(sizeof(struct DADO));
 if (t == NULL)
      return(FALTA_DE_MEMORIA);
 else
     {
      t->info = valor;
      t->elo = p->topo;
      if (p->topo == NULL)
         t->elo = NULL;
      p->topo = t;
      return(SUCESSO);
     }
}

/* ------------------------------------ pop */

int pop (struct PILHA *p, int *info)
{
 struct DADO *t;

 if (p->topo == NULL)
       return(PILHA_VAZIA);
 else
      {
       t = p->topo;
              *info = t->info;
       p->topo = t->elo;
```

```
       free(t);
       return(SUCESSO);
     }
}

/* ---------------------------------- consultar_pilha */

int consulta_pilha (struct PILHA p, int *info)
{
 struct DADO *t;

 if (p.topo == NULL)
     return(PILHA_VAZIA);
 else
     {
      t = p.topo;
      *info = t->info;
      return(SUCESSO);
     }
}

/* ---------------------------------- imprime_erro */

void imprime_erro (int erro)
{
 switch (erro)
 {
  case FALTA_DE_MEMORIA: printf("ERRO: Falta de memória, tecle algo\n");
                                       break;
            case PILHA_VAZIA: printf("ERRO: Pilha vazia, tecle algo\n");
                                     break;
 }
}
```

20.2 - Implementação de uma fila

Programa exemplo (76): O programa permite inserir números inteiros em uma fila. Quando o número digitado for igual à zero (0), todos os números da fila são listados.

```
/* fila.c */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <ctype.h>
#include <string.h>

#define SUCESSO                    1
#define FALTA_DE_MEMORIA           2
#define FILA_VAZIA                 3

void cria_fila (struct FILA *f);
int incluir_fila (struct FILA *f, int valor);
int excluir_fila (struct FILA *f);
```

```
int consultar_fila (struct FILA *f, int *j);
void imprime_erro (int erro);

struct DADO {
                    int info;                 /* informação */
                    struct DADO elo;              / elo para o próximo elemento */
                   };

struct FILA {
                 struct DADO *primeiro;
                 int tamanho;
                 struct DADO *ultimo;
                };

struct FILA *f;

/* ----------------------------------- main */

void main (void)
{
 int valor, erro;
 char ch;

 cria_fila(f);
 do {
      clrscr();
      printf("[I]ncluir, [E]xcluir, [C]onsultar ou [F]im ?");
      do {
            ch = toupper(getche());
          } while (!strchr("IECF", ch));
     if (ch == 'I')
       {
        clrscr();
        printf("Valor: ");
        scanf("%d", &valor);
       }
     switch (ch)
        {
          case 'I': erro = incluir_fila(f, valor);
                       if (erro != SUCESSO)
                          imprime_erro(erro);
                       break;
         case 'E': erro = excluir_fila(f);
                       if (erro != SUCESSO)
                          imprime_erro(erro);
                       break;
        case 'C': erro = consultar_fila(f, &valor);
                       if (erro != SUCESSO)
                          imprime_erro(erro);
                       else
                             {
                               clrscr();
                               printf("Valor: %d\n", valor);
                               getch();
                             }
                      break;
```

```
      }
    } while (ch != 'F');
}

/* ------------------------------------ cria_fila */

void cria_fila (struct FILA *f)
{
 f→primeiro = NULL;
 f→tamanho = 0;
 f→ultimo = NULL;
}

/* ------------------------------------ incluir_fila */

int incluir_fila (struct FILA *f, int valor)
{
 struct DADO *t;

 t = (struct DADO *) malloc(sizeof(struct DADO));
 if (t == NULL)
      return(FALTA_DE_MEMORIA);
 else
    {
      t→info = valor;
      t→elo = NULL;
      if (f→ultimo != NULL)
         f→ultimo→elo = t;
      f→ultimo = t;
      if (f→primeiro == NULL)
         f→primeiro = t;
      f→tamanho = f→tamanho + 1;
      return(SUCESSO);
    }
}

/* ------------------------------------ excluir_fila */

int excluir_fila (struct FILA *f)
{
 struct DADO *t;

 if (f→primeiro == NULL)
       return(FILA_VAZIA);
 else
      {
       t = f→primeiro;
       f→primeiro = t→elo;
       f→tamanho = f→tamanho - 1;
       free(t);
       if (f→primeiro == NULL)
          f→ultimo = NULL;
       return(SUCESSO);
    }
}
```

```
/* ----------------------------------- consultar_fila */

int consultar_fila (struct FILA *f, int *j)
{
 struct DADO *t;

 if (f→primeiro == NULL)
    return(FILA_VAZIA);
 else
    {
     t = f→primeiro;
     *j = t→info;
     return(SUCESSO);
    }
}

/* ----------------------------------- imprime_erro */

void imprime_erro (int erro)
{
 switch (erro)
 {
  case FALTA_DE_MEMORIA:   printf("\nERRO: Falta de memória, tecle algo\n");
                           break;
  case FILA_VAZIA:         printf("\nERRO: Fila vazia, tecle algo\n");
                           break;
 }
 getch();
}
```

20.3 - Lista duplamente encadeada

Uma lista duplamente encadeada possui um elo para o elemento anterior e um elo para o elemento posterior. Possui uma vantagem sobre a lista encadeada, pois este tipo de lista pode ser percorrida em duas direções (para a direita e para a esquerda).

Programa exemplo (77): O programa permite inserir números inteiros em uma lista duplamente encadeada. A inserção pode ser pela **esquerda** ou a **direita**. A exibição dos elementos da lista pode ser feita pela esquerda ou direita.

```
/* dupla.c */

#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <ctype.h>
#include <string.h>

struct ELEMENTO {
                  struct ELEMENTO *anterior;
                  int dado;
                  struct ELEMENTO *posterior;
                 } ;
```

```
struct DESCRITOR {
                              struct ELEMENTO *primeiro;
                              int n;
                              struct ELEMENTO *ultimo;
                             } ;

void inicializa_descritor (struct DESCRITOR *d);
void insere_direita (struct DESCRITOR *d , int n);
void insere_esquerda (struct DESCRITOR *d , int n);
void exibir_lista_direita (struct DESCRITOR d);
void exibir_lista_esquerda (struct DESCRITOR d);

/* ----------------------------------- main */

void main (void)
{
 struct DESCRITOR d;
 int n;
 char op;

 inicializa_descritor(&d);
 do {
      clrscr();
      printf("Número: ");
      scanf("%d", &n);
      if (n != 0)
        {
         printf("Inserir a [E]squerda ou [D]ireita ?");
         do {
               op = toupper(getche());
            } while (!strchr("ED", op));
         switch (op)
         {
           case 'E': insere_esquerda(&d, n);
                          break;
           case 'D': insere_direita(&d, n);
                          break;
         }
        }
    } while (n != 0);
 clrscr();
 do {
       printf("Listar: [E]squerda, [D]ireita ou [F]im?");
      do {
            op = toupper(getch());
           } while (!strchr("EDF", op));
       printf("%c\n", op);
       switch (op)
       {
        case 'E': exibir_lista_esquerda(d);
                       break;
        case 'D': exibir_lista_direita(d);
                       break;
       }
    } while (op != 'F');
```

```
}

/* ---------------------------------- inicializa_descritor */

void inicializa_descritor (struct DESCRITOR *d)
{
 d->primeiro = NULL;
 d->n = 0;
 d->ultimo = NULL;
}

/* ---------------------------------- insere_esquerda */

void insere_esquerda (struct DESCRITOR *d, int n)
{
 struct ELEMENTO *p,*q;

 p = (struct ELEMENTO *) malloc(sizeof(struct ELEMENTO));
 if (p == NULL)
     printf("ERRO: Falta de Memória\n");
 else
     {
      if (d->n == 0)
         {
          p→anterior = NULL;
          p→dado = n;
          p→posterior = NULL;
          d->primeiro = p;
          d->n = 1;
          d->ultimo = p;
         }
      else
          {
           q = d->primeiro;
           p→anterior = NULL;
           p→dado = n;
           p→posterior = q;
           q→anterior = p;
           d->primeiro = p;
           d->n = d->n + 1;
          }
     }
}

/* ---------------------------------- insere_direita */

void insere_direita (struct DESCRITOR *d, int n)
{
 struct ELEMENTO *p,*q;

 p = (struct ELEMENTO *) malloc(sizeof(struct ELEMENTO));
 if (p == NULL)
     printf("ERRO: Falta de memória\n");
 else
     {
      if (d->n == 0)
```

```
         {
          p→anterior = NULL;
          p→dado = n;
          p→posterior = NULL;
          d->primeiro = p;
          d->n = 1;
          d->ultimo = p;
         }
      else
         {
           q = d->ultimo;
           p→anterior = q;
           p→dado = n;
           p→posterior = NULL;
           q→posterior = p;
           d->ultimo = p;
           d->n = d->n + 1;
         }
    }
}

/* ------------------------------------ exibir_lista_direita */

void exibir_lista_direita (struct DESCRITOR d)
{
 struct ELEMENTO *p;

 p = d.ultimo;
 while (p→anterior != NULL)
       {
        printf("Valor: %d\n", p→dado);
        p = p→anterior;
       }
 printf("Valor: %d\n", p→dado);
}

/* ------------------------------------ exibir_lista_esquerda */

void exibir_lista_esquerda (struct DESCRITOR d)
{
 struct ELEMENTO *p;

 p = d.primeiro;
 while (p→posterior != NULL)
       {
        printf("Valor: %d\n", p→dado);
        p = p→posterior;
       }
 printf("Valor: %d\n", p→dado);
}
```

21. Lista de exercícios gerais

✓ Escreva um programa em C que exibe um relógio na tela. O formato do relógio deve ser **hh:mm:ss**.

- ✓ Escreva um programa em C que exibe a data na tela. O formato da data deve ser **dd:mm:aaaa**.